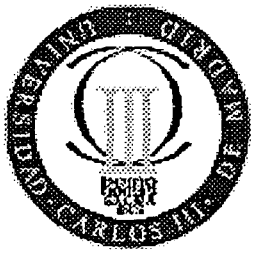
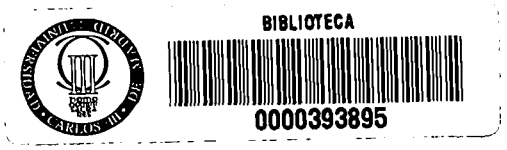


UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Ingeniería en Informática

Proyecto Fin de Carrera

GA-Ensemble: Optimización de conjuntos de clasificadores mediante algoritmos genéticos



Autor: Fco. Javier Ordóñez Morales

Tutor: Prof. Agapito I. Ledezma Espino

Enero, 2007

Tabla de contenidos

Índice de figuras

Índice de tablas

Capítulo 1

INTRODUCCIÓN

Existen muchos ejemplos de cómo una decisión importante no debe tomarse de forma individual (juntas directivas de las empresas, jurados populares en los procesos judiciales, etc.) y debe ser meditada lo suficiente, puesto que debido a su relevancia no es bueno que recaiga bajo la responsabilidad de una sola persona. En este mismo principio se basa la combinación de clasificadores, una de las áreas más interesantes dentro del aprendizaje automático que permite a un sistema tomar decisiones en conjunto valorando la aportación de distintos sistemas de aprendizaje. Este enfoque de combinar diversos clasificadores es conocido como *conjuntos de clasificadores*. La finalidad de los conjuntos es obtener un clasificador más preciso que cualquiera de los clasificadores que forman parte del conjunto.

A la hora de generar los clasificadores que forman parte de un conjunto, se puede utilizar un único algoritmo de aprendizaje o utilizar una combinación de varios algoritmos de aprendizaje. En el primero de los casos se dice que el conjunto de clasificadores generado es un conjunto homogéneo mientras que el segundo caso se dice que el conjunto es heterogéneo.

Una vez que se han generado los clasificadores que formarán parte del conjunto es necesario combinar sus decisiones de alguna manera. Entre las distintas formas de combinar estas decisiones la más común son los votos.

Dentro de los algoritmos de generación de conjuntos homogéneos existen diversos enfoques utilizados para llevar a cabo el proceso de generación los distintos clasificadores. Mientras que algunos de estos métodos utilizan un submuestreo del conjunto de entrenamiento (e.g. *Bagging* [3]), otros manipulan los atributos de entrada para generar diferentes datos de entrenamiento o la salida prevista (e.g. *ECOC* [12]).

Entre los algoritmos de generación de conjuntos heterogéneos destaca el conocido como *Stacking* [37]. Este método aplica diferentes algoritmos de aprendizaje al conjunto de datos con la finalidad de generar los miembros del conjunto. Una vez generados dichos clasificadores, *Stacking* genera un clasificador adicional o metaclasificador para combinar los clasificadores anteriormente generados.

Una de las dificultades en la utilización de *Stacking* es la configuración de sus parámetros de aprendizaje, entre ellos, cuantos y cuales algoritmos de aprendizaje utilizar para construir el conjunto. Este problema es resuelto por el algoritmo *GA-Stacking* [23] mediante la utilización de algoritmos genéticos.

En este Proyecto Fin de Carrera se plantea la optimización de conjuntos de clasificadores heterogéneos basándose en la idea propuesta en *GA-Stacking*. En esta nueva versión, denominada *GA-Ensemble* los clasificadores se generan en una fase previa y luego son combinados mediante voto o metaclasificadores según determine un algoritmo genético.

La estructura del resto del documento se detalla a continuación. En el Capítulo 2 se presenta una panorámica general de la cuestión en cuanto a aprendizaje automático y, en especial, conjuntos de clasificadores se refiere. En el Capítulo 3 se enumeran los objetivos de este Proyecto Fin de Carrera, tanto el objetivo general como los específicos. En el Capítulo 4 se detalla el trabajo realizado ofreciendo una descripción desde un alto nivel hasta llegar a los detalles. En el Capítulo 5 se muestran los

resultados obtenidos en la fase experimental de este proyecto. En el Capítulo 6 se muestran las conclusiones obtenidas con la realización del proyecto. Por último, en el Capítulo 7 figuran los trabajos futuros que se pueden llevar a cabo sobre la base del trabajo realizado.

Capítulo 2

ESTADO DE LA CUESTIÓN

En este apartado se ofrece una visión general del área en el que se enmarca este Proyecto Fin de Carrera. De forma introductoria en el primer apartado se hace una breve descripción del concepto de Aprendizaje Automático, definiendo de forma concisa los diferentes tipos que existen. Posteriormente, en la sección 2.2 se desarrolla el concepto de conjuntos de clasificadores tomando en cuenta aspectos como su definición, eficiencia y técnicas de construcción. En el apartado 2.3 se presenta y define la técnica de generación de conjuntos heterogéneos conocida como *Stacked Generalization*, base de este trabajo. En la sección 2.4 se hace una introducción a los algoritmos genéticos. En la sección 2.5 se describe *GA-Stacking*, el sistema que sirve de base a este proyecto. Por último en la sección 2.6 se incluye la descripción de las herramientas que se han utilizado para llevar a cabo este Proyecto Fin de Carrera.

2.1 Aprendizaje Automático

El Aprendizaje Automático es un amplio campo de la Inteligencia Artificial que trata del desarrollo de algoritmos y técnicas que permitan "aprender" a los programas, es decir, que permita a estos mejorar automáticamente usando para ello la experiencia. Este campo toma conceptos de otros campos como la estadística, la biología, o la teoría de la información.

El Aprendizaje Automático tiene una amplia variedad de aplicaciones, entre las que se pueden mencionar el procesamiento del lenguaje natural, motores de búsqueda, diagnóstico medico, bioinformática, detección de fraude en el uso de tarjetas de crédito, clasificación de secuencias de ADN, reconocimiento de objetos en visión por ordenador, etc.

Según Mitchell [28] "un programa de ordenador se dice que aprende de la experiencia E con respecto a una cierta clase de tarea T y medida de funcionamiento P, si su funcionamiento en la tarea T según lo medido por P, mejora con la experiencia E."

De acuerdo al tipo de aprendizaje, existen tres grupos de técnicas: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

2.1.1 Aprendizaje Supervisado

El aprendizaje supervisado es una rama del aprendizaje automático donde el algoritmo produce una función que establece una correspondencia entre la serie de datos de entrada que recibe y las salidas deseadas del sistema. Estos datos de entrada o ejemplos de entrenamiento (típicamente vectores) poseen una etiqueta o clase asociada, es decir, han sido clasificados previamente. De esta forma durante el proceso de aprendizaje, el algoritmo compara su salida actual con la etiqueta de clase para comprobar que dicho proceso se ha llevado a cabo correctamente, o en caso contrario, realizar los cambios que sean necesarios.

Cada ejemplo (llamado también instancia) dentro del conjunto de entrenamiento se puede expresar mediante la forma atributo-valor o mediante relaciones. Cuando a un programa de aprendizaje se le pasa un conjunto de ejemplos $\{(x_1,y_m),\ldots,(x_m,y_m)\}$ para describir una función desconocida $y = f(x)$, los valores de x_i son vectores de la forma

$$(x_{i,1}, x_{i,2}, \ldots, x_{i,n}, y)$$

donde $(x_{i,j})$ se refiere a cada característica (atributo) de (x_i), y se refiere a la clase de la instancia y n es el número total de atributos de la instancia. Los atributos o características que forman parte de una instancia pueden ser nominales y numéricos. Un ejemplo de atributo nominal es el género, siendo sus posibles valores *Masculino* o

Femenino. Sin embargo los valores de atributos como *Tamaño* o *Peso* pueden ser numéricos y, en consecuencia, pueden ser llamados continuos.

Igualmente los valores de y pueden ser también nominales o numéricos. En caso de que estos valores pertenezcan a un número definido de clases $\{1,...,K\}$ se dice que es una tarea de *clasificación* y en caso de y sea numérico la tarea es de *regresión*.

El conjunto de todos los posibles valores que puede tomar los atributos de x se conoce como espacio de instancias o espacio de entrada. El conjunto de los posibles valores de y se conoce como espacio de salida.

Las aplicaciones del aprendizaje supervisado son variadas y numerosas, como por ejemplo, su aplicación con éxito en campos tan diversos como la bioinformática, los sistemas de recuperación de datos, reconocimiento del habla, reconocimiento de patrones y detección de *spams*.

Generalmente, cuando se tiene que llevar a cabo una tarea de clasificación o de regresión, se utiliza un conjunto de instancias o ejemplos para que el algoritmo de aprendizaje construya un *clasificador*. Mientras que el conjunto de ejemplos utilizados para el aprendizaje es llamado conjunto de *entrenamiento* o *aprendizaje*, el conjunto de instancias que no se ha utilizado para construir el clasificador y que se usa para validarlo se llama conjunto de *prueba* o *test*.

Usando el conjunto de prueba se evalúa la precisión del clasificador. Basándose en los ejemplos del conjunto de *test* que el clasificador ha clasificado correctamente, se obtiene una *precisión de clasificación*, normalmente, observando qué porcentaje de instancias han sido clasificadas correctamente. La tarea de clasificación se puede definir como tomar como entrada una nueva instancia sin clase conocida, y usando un clasificador o método, obtener la clase de dicho ejemplo.

Dentro del aprendizaje supervisado, de acuerdo al tipo de representación de los datos de entrada, se puede hacer una clasificación en dos grupos: los representados en la forma atributo-valor y los que están representados en forma de relaciones (lógica de primer orden).

El grupo de algoritmos que utilizan la representación de atributo-valor se divide a su vez en otros dos grupos: algoritmos subsimbólicos y simbólicos. Entre los algoritmos subsimbólicos destacan las redes de neuronas [17] y los algoritmos genéticos [18] (cuando son usados como técnica de clasificación) mientras que dentro del aprendizaje simbólico destacan los árboles de decisión (e.g. *ID3* [29], *C4.5* [30]) y los sistemas basados en reglas (e.g. *AQ* [27], *PART* [14])

2.1.2 Aprendizaje No Supervisado

Este tipo de aprendizaje se diferencia del aprendizaje supervisado en que no se conoce "a priori" el atributo dependiente. No es fácil ver como con esta técnica es posible realizar un aprendizaje dado que no se obtiene ningún *feedback* del entorno, sin embargo es posible definir un marco formal para el aprendizaje no supervisado basado en la idea de que el objetivo es construir representaciones de las entradas que puedan ser usadas para la toma de decisiones o para predecir futuros ejemplos.

Ejemplos de este tipo de aprendizaje son los algoritmos de agrupamiento o *clustering*, entre los se destacan *EM* [9], *K-Medias* [26] y los *Sistemas Clasificadores* [25]. El aprendizaje no supervisado es también útil para la compresión de datos, ya que todos los algoritmos de compresión se basan, explicita o implícitamente, en una distribución de probabilidad sobre un conjunto de entradas.

2.1.3 Aprendizaje por Refuerzo

En el aprendizaje por refuerzo el algoritmo aprende observando el mundo que le rodea. En este tipo de aprendizaje el algoritmo utilizado recibe las entradas y una evaluación o *feedback*. Es por ello que el algoritmo debe aprender que acción es la que ofrece mayor rendimiento a largo plazo, puesto que el futuro *feedback* que reciba vendrá como respuesta a sus acciones. Entre los algoritmos de aprendizaje por refuerzo destacan *Q-Learning* [35] y *ARTDP* (*Adaptative Real Time Dynamic Programming*) [2].

2.2 Conjuntos de clasificadores

Según Dietterich [10], "un conjunto de clasificadores es un grupo de clasificadores cuyas decisiones individuales se combinan de alguna manera (típicamente mediante votos con peso o sin peso) con el objetivo de clasificar nuevos ejemplos". Dentro del aprendizaje supervisado, una de las áreas de investigación más activas es el estudio de métodos para construir buenos conjuntos de clasificadores. Generalmente los conjuntos de clasificadores mejoran la precisión de cualquiera de los clasificadores individuales que forme parte de éste [4,12].

Para que un conjunto de clasificadores sea más preciso que cualquiera de sus miembros, es necesaria y suficiente la condición de que estos sean a su vez diversos y precisos. Un clasificador se considera preciso si comete un error menor que el que se podría obtener eligiendo aleatoriamente una clase entre las clases disponibles. Del mismo modo, se considera que dos clasificadores son diversos, si los errores que cometen sobre los datos de entrada no están correlados.

Para ver porqué la precisión y la diversidad son necesarias y suficientes se plantea un esquema en el que existe un conjunto de tres clasificadores, h_1, h_2 y h_3, los cuales deben clasificar una nueva instancia x. En el caso de que los tres clasificadores no sean diversos, es decir, sean idénticos, se da el caso de que si $h_1(x)$ es erróneo, $h_2(x)$ y $h_3(x)$ también serán erróneos. Sin embargo, si los errores que cometen sobre los datos de entrada no son los mismos, es decir, no están correlados, en el caso que $h_1(x)$ esté errado, $h_2(x)$ y $h_3(x)$ no tendrían porque estar mal y podrían ser correctos, en cuyo caso utilizando el voto mayoritario, la instancia x sería clasificada correctamente. De una forma más precisa, si la tasa de error de L hipótesis h_l son todas iguales a $p < ½$ y los errores que comenten son independientes, la probabilidad de que utilizando el voto mayoritario para combinar las decisiones de los clasificadores esté errada, viene dada por el área bajo la curva de una distribución binomial en donde más de $L/2$ hipótesis estén erradas. En la Figura 2.1 se muestra un hipotético conjunto formado por 21 hipótesis, cada una de ellas con una tasa de error del 0.3. El área bajo la curva en donde 11 o más hipótesis estén simultáneamente erradas es 0.026, que es mucho menor que la tasa de error individual de las hipótesis [10].

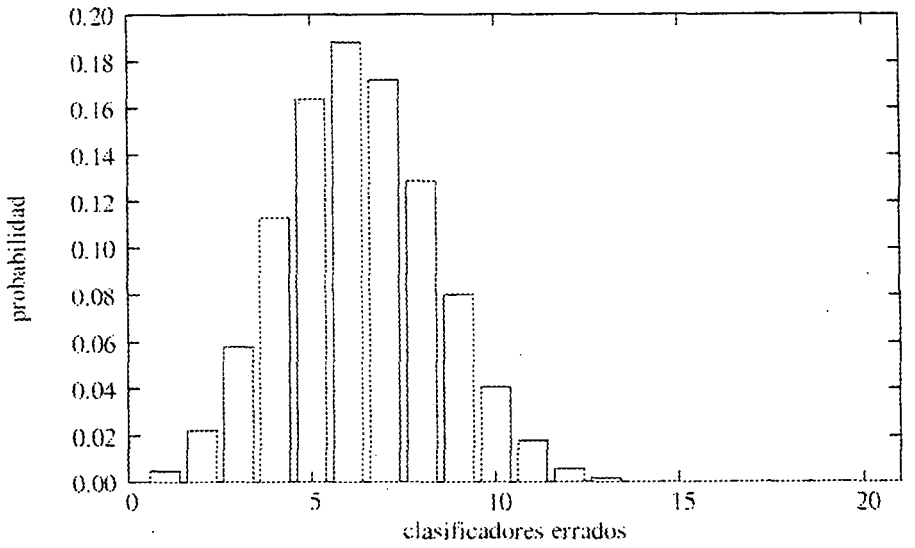


Figura 2.1: Probabilidad de que exactamente l (de 21) hipótesis comenta un error, asumiendo que cada hipótesis tiene una tasa de error de 0,3 y cometen sus errores independientemente de las demás hipótesis.

2.2.1 Porqué funcionan los conjuntos de clasificadores

Según Dietterich [10], existen tres razones fundamentales por las cuales se pueden encontrar buenos conjuntos de clasificadores.

- ✓ Razón estadística: Un algoritmo de aprendizaje se puede ver como una búsqueda en un espacio de hipótesis con el objetivo de identificar la mejor hipótesis en dicho espacio. El problema estadístico se presenta cuando el conjunto de datos que se posee es demasiado pequeño en comparación con el espacio de hipótesis. Sin los datos suficientes, el algoritmo de aprendizaje puede encontrar una gran cantidad de hipótesis dentro del espacio de hipótesis con la misma precisión sobre los datos disponibles. Si estos clasificadores se combinan mediante un conjunto de clasificadores, disminuye el riesgo de seleccionar un clasificador que devuelva una hipótesis errónea. En la Figura 2.2 (parte superior izquierda) se representa gráficamente esta situación.

- ✓ Razón computacional: Muchos algoritmos de aprendizaje funcionan llevando a cabo búsquedas locales que pueden quedar atrapadas en máximos locales. En los casos en los que los datos de entrenamiento son suficientes, y por lo tanto no

existe problema estadístico, se puede presentar el problema de tipo computacional en el que sea muy difícil para el algoritmo de aprendizaje encontrar las mejores hipótesis. Por ejemplo, el entrenamiento óptimo para las redes de neuronas y los árboles de decisión es un problema *NP-completo*. Si se llevan a cabo una serie de búsquedas locales con puntos de partida diferentes para obtener hipótesis que luego se combinan, se puede obtener una mejor aproximación a la hipótesis real, en vez de utilizar cualquiera de las hipótesis encontradas (Figura 2.2 parte superior derecha).

- ✓ Razón representacional: En la mayoría de las aplicaciones de aprendizaje automático la función real, *f*, puede no ser representada por ninguna de las hipótesis del conjunto de hipótesis. Al combinar varias hipótesis, es posible que se aumente el espacio de posibles funciones representables y con ellas las hipótesis que se pueden representar y, de esta manera, se podría aproximar mejor la función real (*f*) (Figura 2.2 parte inferior).

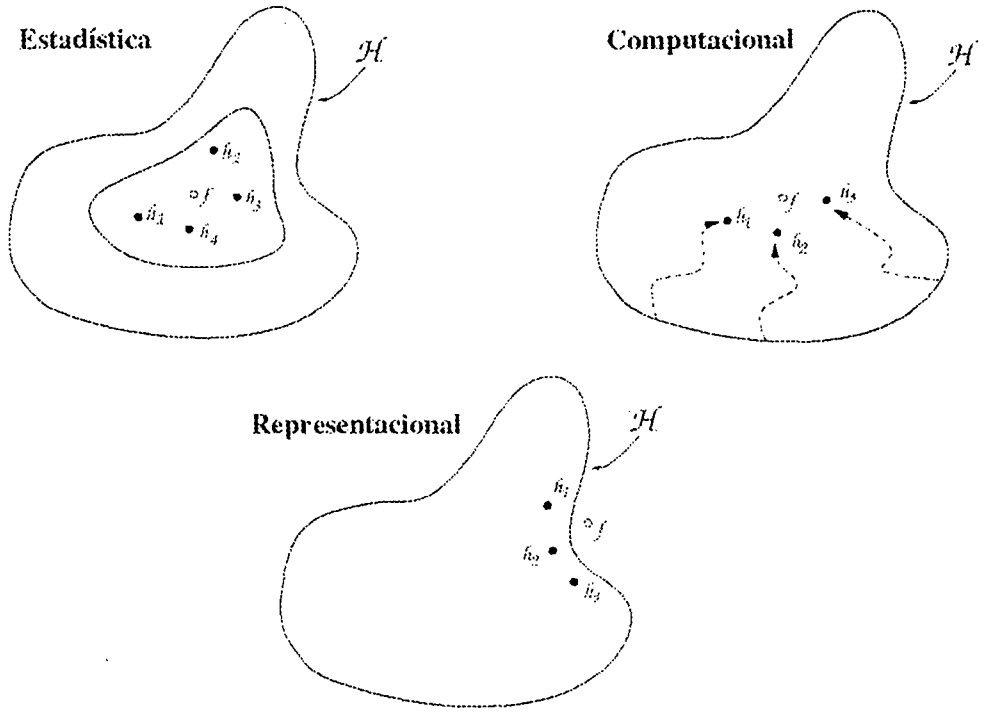


Figura 2.2: Razones fundamentales por las que un conjunto de clasificadores puede funcionar [11]

2.2.2 Métodos de construcción de conjuntos de clasificadores

Actualmente han sido desarrollados muchos métodos para generar conjuntos de clasificadores. Tomando en consideración aquellos métodos que se pueden aplicar a una gran cantidad de algoritmos, Dietterich [10] los clasifica en:

- ✓ *Voto bayesiano*: *enumeración de hipótesis*. Tomando como base el teorema de Bayes se consideran todas las hipótesis en el espacio de hipótesis como parte del conjunto, teniendo cada una un peso asociado equivalente a su probabilidad posterior. Este método es aplicable en tareas de aprendizaje no muy complejas, donde se pueden enumerar todas las hipótesis del espacio de hipótesis y calcular su probabilidad "a posteriori".

- ✓ *Manipulación de los ejemplos de entrenamiento*. Una segunda manera de generar conjuntos de clasificadores, es a partir de la manipulación de los ejemplos de entrenamiento con la idea de generar múltiples hipótesis. El algoritmo de aprendizaje, cualquiera que sea, se ejecuta repetidas veces utilizando un conjunto distinto de instancias de entrenamiento cada vez, generando de este modo los clasificadores que forman parte del conjunto. Esta técnica funciona especialmente bien con algoritmos de aprendizaje *inestables*, es decir, aquellos cuyo modelo resultante varía mucho en respuesta a pequeños cambios en los ejemplos de entrenamiento. Por ejemplo, los árboles de decisión, las redes de neuronas artificiales y los algoritmos de aprendizaje basados en reglas son algoritmos inestables. Por el contrario, los métodos de regresión lineal y el vecino más cercano, suelen ser muy estables. *Bagging*[3] es considerado como el método más sencillo para manipular los datos de entrenamiento. Esta técnica funciona generando múltiples versiones de un clasificador concreto y combinando las decisiones de estas diferentes versiones en una sola predicción por medio del mecanismo de voto mayoritario (la clase que obtiene más votos por parte de los clasificadores, es la clase ganadora). La formación de las diferentes versiones se basa en el submuestreo con reemplazo del conjunto de entrenamiento para generar un grupo diferente de hipótesis, utilizando cada muestra obtenida como conjunto de entrenamiento. Otro algoritmo ampliamente

utilizado y con distintas versiones es *Boosting* [15]. *Boosting* esta basado en la asignación de pesos a las instancias mal clasificadas por los clasificadores anteriores en un proceso de generación de clasificadores secuencial.

- ✓ *Manipulación de los atributos de entrada.* La tercera forma de generar conjuntos, según Dietterich, es mediante la manipulación de los atributos de entrada disponibles a la hora de utilizar al algoritmo de aprendizaje. Una debilidad de este tipo de técnica es que sólo funciona cuando los atributos de entrada son altamente redundantes. Un ejemplo de este tipo de técnica es el aplicado por Cherkauer [6] en donde lleva a cabo diferentes agrupaciones de losa tributos de entrada para generar los clasificadores que forman parte del conjunto.

- ✓ *Manipulación de las salidas.* Otra técnica para generar conjuntos de clasificadores es la manipulación de la salida esperada, es decir, la clase de la instancias (y). Uno de los métodos más representativos de estas técnicas es el conocido como *ECOC* [12]. Este método asume que el número de clases, K, es grande. De esta forma se crean nuevas tareas de aprendizaje dividiendo aleatoriamente las K clases en dos subconjuntos A_l y B_l. Los datos de entrada pueden ser entonces re-etiquetados de forma tal que todas las instancias en el conjunto A_l de cualquiera de las clases originales sean re-etiquetadas con la etiqueta 0 y todas las instancias de cualquier clase en B_l son re-etiquetadas con la etiqueta 1. Con estos datos se entrena el algoritmo de aprendizaje, generando un clasificador h_l. Al repetir este proceso L veces se obtiene un conjunto formado por L clasificadores (h_1,..., h_l). Una vez creado el conjunto, *ECOC* clasifica una nueva instancia x aplicando cada clasificador h_l a ésta. Si $h_l(x) = 0$, entonces cada clase en A_l recibe un voto, pero si $h_l(x) = 1$, entonces cada clase en B_l es la que recibe un voto. Una vez que los L clasificadores han votado, la clase mayoritaria, es decir, la que tenga mayor número de votos será seleccionada como la predicción del conjunto.

- ✓ Introducción de aleatoriedad. Otra forma de generar conjuntos, según Dietterich, para la generación de conjuntos es la incorporación de aleatoriedad dentro del algoritmo de aprendizaje. Por ejemplo, realizando el entrenamiento de una red

de neuronas con el mismo conjunto de entrenamiento, pero con diferentes pesos iniciales seleccionados de forma aleatoria, es posible obtener clasificadores bastante distintos.

La clasificación propuesta anteriormente por Dietterich [10] sólo considera los conjuntos de clasificadores que se forman a partir de un único algoritmo de aprendizaje, es decir, que los clasificadores generados son homogéneos. Pero, como se ha descrito anteriormente, existe otra manera de generar conjuntos de clasificadores mediante la aplicación de algoritmos de aprendizaje diferentes, es decir, mediante el uso de clasificadores heterogéneos. Una de las ventajas de este tipo de conjuntos de clasificadores es que haciendo uso de clasificadores heterogéneos el conjunto generado aprovecha los distintos *bias* o sesgo de los algoritmos utilizados.

2.3 Stacking

Stacking es la abreviación de *Stacked Generalization* [37]. Junto con *Bagging* y *Boosting* es la técnica de generación de conjuntos de clasificadores más utilizada, pero a diferencia de estas, valida diferentes algoritmos de aprendizaje para generar el conjunto de clasificadores, es decir, genera un conjunto de clasificadores heterogéneos. Los algoritmos de aprendizaje que puede usar *Stacking* son tan diversos como: los árboles de decisión, los algoritmos basados en instancias, las redes de neuronas, etc. Cada uno de esos algoritmos de aprendizaje utiliza una representación diferente del conocimiento y distintos *bias* de aprendizaje, por lo que el espacio de hipótesis es explorado de forma diferente. Es por esto que se espera que los clasificadores generados no estén correlados y, que combinando los diferentes clasificadores se obtengan predicciones más precisas que usando cualquiera de los algoritmos de aprendizaje individuales que forman parte del conjunto.

Una vez que los clasificadores han sido generados *Stacking* los combina usando un clasificador de nivel superior o meta-clasificador. No se usa un mecanismo de votos, a diferencia de *Bagging* y *Boosting*, porque si se diese el caso de que la mayoría de los clasificadores llevasen a cabo malas predicciones, esto conduciría a una clasificación final errónea. El meta-clasificador (o modelo de nivel-1) generado es utilizado para

modelizar cómo se deben combinar las decisiones de los clasificadores base (o modelos de nivel-0). En la Figura 2.3 se muestra el funcionamiento general de *Stacking*.

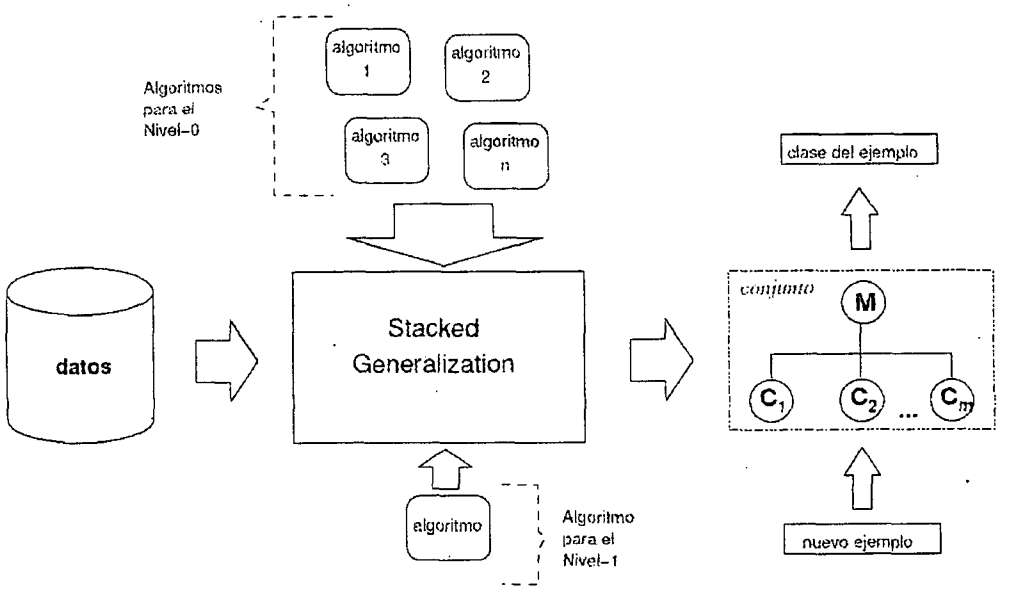


Figura 2.3: Funcionamiento general de Stacking

Formalmente, dado un conjunto de datos S, *Stacking* genera, primeramente, un subgrupo de conjuntos de entrenamiento $S_1, \ldots, S_T$ para después seguir un proceso similar al de la validación cruzada, es decir, deja uno de los subconjuntos (e.g. S_j) fuera para utilizarlo posteriormente para entrenar el meta-clasificador. El resto de las instancias $S^{(-j)} = S - S_j$ son utilizadas para generar los clasificadores de nivel-0 mediante la aplicación de K algoritmos de aprendizaje distintos, $k = 1,\ldots,K$, y de este modo obtener K clasificadores. Una vez que los modelos de nivel-0 han sido generados, el conjunto S_j, que había dejado fuera, es usado para entrenar el meta-clasificador (clasificador de nivel-1). Los datos de entrenamiento de nivel-1 se forman a partir de las predicciones de los modelos de nivel-0 sobre las instancias en S_j, las cuales han sido reservadas para este propósito. Los datos de nivel-1 tienen K atributos cuyos valores son las predicciones de cada uno de los K clasificadores de nivel-0 para cada instancia en S_j. De este modo, una instancia de entrenamiento de nivel-1 está constituida por K atributos (las K predicciones) y la clase objetivo, la cual es la clase real para cada instancia en particular en S_j. Una vez que los datos del nivel-1 han sido construidos a partir de todas las instancias en S_j, cualquier algoritmo de aprendizaje puede ser utilizado para generar el modelo de nivel-1. Para completar el proceso, los modelos de nivel-0 son regenerados

a partir del conjunto *S* completo (de esta manera se espera que los clasificadores sean ligeramente más precisos). Para clasificar una nueva instancia, los modelos de nivel-0 producen un vector de predicciones que es la entrada al modelo de nivel-1, el cual genera la predicción final del conjunto.

2.3.1 Variantes de Stacking

Hay una serie de trabajos relacionados con *Stacking* que se pueden considerar variantes de este enfoque o bien que mantienen muchas similitudes. A continuación se mostrará un resumen de algunos trabajos representativos basados en *Stacking* u otros algoritmos similares relacionados con éste.

Chan y Stolfo [5] propusieron un algoritmo muy parecido a *Stacking* al cual bautizaron como *combinador*. También propusieron una variante de éste a la que llamaron *combinador-de-atributos*, en la cual, los atributos del meta-nivel están formados, no solo por las predicciones de clase sino también por los atributos originales de la instancia. Sin embargo, como han demostrado otros estudios (Schaffer [31]) esto puede llevar a un peor rendimiento del conjunto.

Por otra parte Chan y Stolfo [5] proponen un enfoque que utiliza lo que denominan un "árbitro". Siendo el "árbitro" un clasificador individual independiente del resto de los clasificadores base, que es entrenado sobre un subconjunto del conjunto original de datos, el cuál está formado por las instancias en las que los clasificadores base están en desacuerdo. El propósito de el "árbitro" es ofrecer una predicción alternativa y más elaborada cuando los clasificadores base presentan contradicciones.

Otros autores, como Ting [33] proponen la utilización de las predicciones de los clasificadores base para aprender una función que refleje la medida interna de confianza del algoritmo en una estimación de su precisión sobre la salida. Esta función puede ser utilizada para combinar los conocimientos del clasificador.

Todorovski y Džeroski [34] desarrollaron una variante de *Stacking* que predice qué clasificador es el más indicado dado un ejemplo específico. Esta variante usa un nuevo método de aprendizaje en el meta-nivel. Este método, conocido como *meta*

árboles de decisión (MDT), sustituye las predicciones clase-valor en sus nodos hoja por los clasificadores de nivel-base. En esta variante los meta-datos están compuestos de propiedades de las distribuciones de probabilidades que reflejan la confianza de los clasificadores de nivel-base, como la entropía y la probabilidad máxima, en vez de las propias distribuciones.

2.4. Algoritmos Genéticos

Los Algoritmos Genéticos (AG's) son una técnica de la Inteligencia Artificial que intenta imitar la evolución biológica. Los AG's son procesos de búsqueda basados en la teoría de la evolución de Darwin, apropiados para resolver problemas donde el espacio de soluciones puede resultar demasiado extenso. Estos algoritmos buscan generar un conjunto de soluciones para un problema basados en un nivel de aptitud (*fitness*), la eficacia del algoritmo será plenamente dependiente de los criterios que se consideren para la determinación del *fitness* así como de su representación. Los algoritmos genéticos forman parte de una familia denominada algoritmos evolutivos, que incluye, entre otras técnicas, las estrategias de evolución, la programación evolutiva y la programación genética.

Usando términos de búsqueda clásica un algoritmo genético es una variante de la búsqueda de haz. Un algoritmo genético puede descomponerse en tres elementos:

- ✓ La población, o lo que en búsqueda clásica sería el haz. Esta población esta compuesta por diversas soluciones candidatas o individuos y conforma el espacio de búsqueda del algoritmo. Para codificar estos individuos o soluciones candidatas se pueden usar diversas formas de representación, por ejemplo mediante codificación binaria, decimal o hexadecimal, aunque lo más común es que sean representados mediante cadenas de bits. Esta representación, al ser independiente del dominio, hace que los AG's sean muy flexibles y potentes.

- ✓ Operadores de búsqueda. Estos operadores se definen como las operaciones que permiten transformar una solución candidata actual en otra. Permiten que a partir de un conjunto de soluciones o población se pase a la siguiente. Al operar sobre cadenas de bits son independientes del dominio. Los tres operadores más

habituales son aquellos en los que se puede ver una mayor analogía biológica, es decir, aquellos que definen principalmente la evolución natural:

- Reproducción: un individuo o solución candidata se copia a la siguiente generación sin sufrir ninguna modificación.
- Cruzamiento: tomando como base dos individuos, estos se cruzan dando lugar a otros nuevos individuos, diferentes de sus padres, que serán los que formen parte de la siguiente generación de la población. Existen diversas variaciones de este operador (principalmente, de un punto y de dos puntos).
- Mutación: un individuo o solución candidata se ve modificado tan sólo en un bit de su cromosoma (pudiendo mutar de 0 a 1 ó de 1 a 0). El bit que se muta es seleccionado de forma aleatoria de entre los bits que componen el individuo, con una cierta probabilidad determinada a priori, aunque generalmente es del 1%.

✓ La función heurística (o función de *fitness*). Esta función mide la aptitud o lo adecuada que es una solución candidata. Lo que busca un AG son soluciones candidatas que maximicen o minimicen esta función.

Los AG's parten de una población inicial normalmente generada de forma aleatoria. A partir de esta primera población comienza un proceso iterativo. Primeramente se evalúan las soluciones candidatas actuales (de acuerdo con la función heurística), y sobre estas soluciones se aplican los operadores genéticos hasta que es obtenida una nueva población (nueva generación). Este proceso de generar nuevas poblaciones continúa hasta que se encuentra un individuo que se considera lo suficientemente bueno, el algoritmo llega a un punto donde es incapaz de encontrar mejores individuos o se alcanza el número de generaciones predefinidas.

Un AG básico se basa en el siguiente pseudocódigo:

1. Elección un población inicial, normalmente de forma aleatoria.
2. Evaluación de la adecuación de los individuos de la población
3. Repetir

Seleccionar los individuos mejor adaptados para su reproducción

Crear la nueva generación usando cruzamiento y mutación (operadores genéticos)

Evaluar los nuevos individuos generados

Reemplazar la parte menos adecuada de la población con los nuevos individuos

4. Hasta condición de terminación

La producción de una nueva generación a partir de la generación anterior (pasos 3.1, 3.2, 3.3 y 3.4) se describe a continuación. Primero se seleccionan los individuos mejor adaptados y con ellos se forma una nueva población, denominada población auxiliar. Para poder crear la generación siguiente, de entre los individuos de esta población auxiliar, se seleccionan las soluciones candidatas estocásticamente. Normalmente, para una solución candidata, la probabilidad de ser seleccionada es el cociente entre su *fitness* y el *fitness* total de la población. Esto implica que existirán varias copias de individuos muy buenos en la siguiente generación, pero que probablemente no existan individuos cuya función de *fitness* sea pobre. Aunque debido a la aleatoriedad, un individuo siempre tiene alguna posibilidad de aparecer en la siguiente generación, aunque sea considerado un individuo malo. Este método es conocido como "selección proporcional al *fitness*", aunque existen otros métodos, como el torneo o el ranking.

Para obtener la siguiente generación se aplican los operadores genéticos a un porcentaje de la población seleccionado aleatoriamente. En el caso del sobrecruzamiento se toman dos padres y se producen dos descendientes, por lo que se lleva a cabo un número de veces igual a la mitad del tamaño de la población. Sin embargo, la mutación se aplica a un porcentaje fijo de individuos.

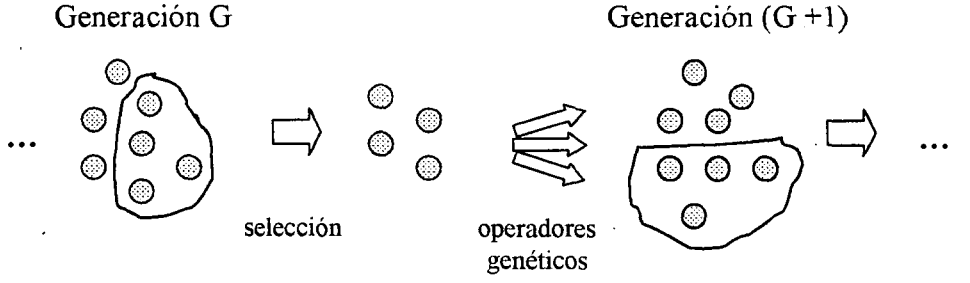


Figura 2.4: Proceso general de los Algoritmos Genéticos

En la Figura 2.4 se muestra gráficamente el proceso seguido por los Algoritmos Genéticos.

2.4.1 Optimización mediante AG's

A lo largo de las últimas décadas han ido apareciendo un conjunto de técnicas que comparten un principio común, el de emular los principios de la evolución natural para la optimización de problemas. De entre todas estás técnicas que han mostrado ser útiles y potentes, las más asentadas y aceptadas por la comunidad científica son los *Algoritmos Genéticos*. Los AG's, a pesar de ser una herramienta novedosa, han demostrado ser herramientas muy potentes por la cantidad de problemas que son capaces de resolver y por la capacidad que tienen para ser aplicados a multitud de dominios. Esto se traduce en que estas técnicas han sido y son ampliamente utilizadas en la industria en campos como el diseño de circuitos, distribución de componentes en la superficie de una antena, optimización del recorrido de tuberías en un edificio, diseño de la forma del ala de un avión supersónico o en la optimización de la situación de órbitas de satélites, proporcionando resultados tan buenos que han sorprendido a los propios expertos.

Desde el punto de vista más formal, son la única de estas técnicas que posee una base matemática, que aunque ha sido discutida en numerosas ocasiones por ciertos sectores puristas, no ha sido rebatida y es aceptada. Se trata del problema conocido como *two-armed-bandir* [19].

En cuanto a la utilidad de los AG's en el área de conjuntos de clasificadores, recientemente Zhou [38] determinó cual es un número adecuado de clasificadores que deben formar parte de un conjunto para llevar a cabo tareas de optimización usando AG's. Otro ejemplo es el algoritmo *GA-Stacking*[23], el cual describe un nuevo enfoque, basado en AG's, que busca obtener la configuración óptima de los parámetros de *Stacking* para un problema dado, y que posteriormente se explicará con más detalle dentro de este Proyecto Fin de Carrera.

2.4.2 Problemas de los AG's

Pese a que los algoritmos genéticos son una técnica que se ha demostrado eficaz en la resolución de un gran número de problemas, también es cierto que plantea una serie de interrogantes.

En principio, es necesario realizar un cierto número de iteraciones para que la aplicación de un algoritmo genético pueda resultar útil. Sin embargo esto entraña un problema, "¿cuántas generaciones hacen falta?". A priori se podría considerar que a mayor número de generaciones mayor probabilidad de encontrar la solución óptima al problema, pero en determinados problemas esto no es así y aumentar el número de generaciones no mejora la calidad de las soluciones encontradas. También se debe tener en cuenta que en ciertos problemas el tiempo para generar una solución es limitado, y por tanto el número de generaciones que se pueden llevar a cabo.

Otro problema que se presenta al utilizar algoritmos genéticos, y quizás el más difícil de resolver, reside en encontrar una función de adecuación (*fitness*) que permita valorar las soluciones encontradas de manera objetiva. El conocer cómo de buena o mala es una solución y codificar este conocimiento en un algoritmo no es trivial, ya que requiere que el ingeniero responsable de la codificación del algoritmo cuente con un experto en el dominio del problema.

Cuando se trabaja con algoritmos genéticos la aleatoriedad está muy presente y en muchos casos está más que justificada su utilización, sin embargo también trae consigo inconvenientes. Hay que destacar que el emplear aleatoriedad no es malo de por sí, ya que en algunos problemas puede provocar que se recupere de un mínimo local y aproximarse más a la solución óptima. Sin embargo, al contar con aleatoriedad, el proceso de resolución de un problema hace imposible, o extremadamente difícil, conseguir obtener dos ejecuciones iguales. Esto implica que para un mismo problema se va a poder generar soluciones diferentes y en ocasiones esto puede hacer descartar la aplicación de esta técnica.

2.5 GA-Stacking

La tesis doctoral de Agapito Ledezma [23] es el punto de partida de este proyecto. Parte de esta tesis consistió en el desarrollo de un sistema que, valiéndose de algoritmos evolutivos, generaba configuraciones óptimas del algoritmo *Stacking* para dominios concretos. Este sistema, denominado *GA-Stacking*, consiguió obtener buenos resultados para los dominios en los que se aplicó. En el presente apartado se procederá a dar una breve descripción del diseño de *GA-Stacking* para así poder entender mejor las bases de este trabajo.

2.5.1 Motivación

Uno de los problemas a los que se enfrenta *Stacking*, es que el conjunto de clasificadores que genera está compuesto por un grupo de modelos creados a partir de distintos algoritmos de aprendizaje. Por lo que, normalmente, es difícil identificar de forma clara que algoritmos se deben usar para generar los modelos de nivel-0 y que algoritmo debe utilizarse para generar el modelo de nivel-1. En principio, cualquier algoritmo puede ser usado para generar los modelos. *GA-Stacking*, sin embargo, va más allá, basándose en la idea de usar distribuciones de probabilidades de clase como datos de meta-nivel propuesta por Ting y Witten [33], busca obtener la configuración óptima de los parámetros de *Stacking* utilizando algoritmos genéticos.

El término de *GA-Stacking* es el acrónimo en inglés de *Genetic Algorithms for Stacking*. *GA-Stacking* considera los parámetros de *Stacking* como un problema de optimización, abordando dicho problema mediante AG's.

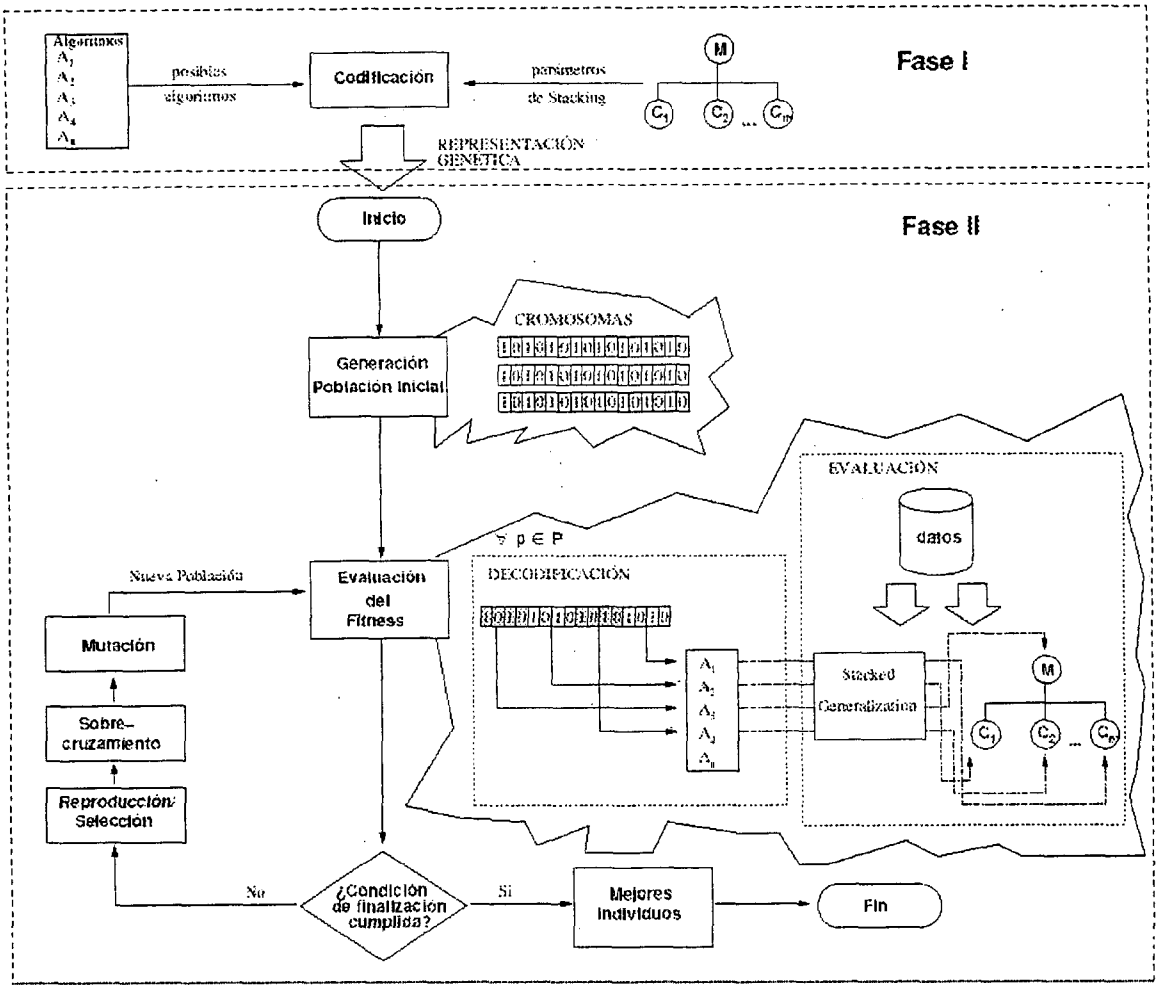


Figura 2.5: Marco de GA-Stacking

2.5.2 Codificación de las soluciones

El proceso de codificación de las soluciones se produce en lo que se puede considerar una fase previa (Figura 2.5, Fase I). Existen diversas maneras de representar las soluciones de un problema para que éste pueda ser tratado mediante la aplicación de algoritmos genéticos. Para representar las posibles soluciones o individuos en el enfoque que *GA-Stacking* propone, se hizo uso de una representación binaria ya que esta permite el empleo de AG's canónicos. Los AG's canónicos son la forma original propuesta por Holland [18], y en donde los operadores genéticos tienen un carácter completamente general y la base matemática es más rigurosa. Además, existe un amplio estudio de la capacidad de barrido del espacio de búsqueda que no está contrastado cuando la base de codificación deja de ser binaria.

Respecto al tamaño del cromosoma que representa al individuo, este se encuentra determinado por dos factores:

- ✓ el número de algoritmos, *m*, que pueden ser seleccionados para generar los clasificadores, tanto los de nivel-base como el del meta-nivel.
- ✓ el número máximo de clasificadores de nivel-base, *n*, que pueden formar parte del conjunto

Si en la codificación de los individuos sólo se considera el nombre del algoritmo que se usa para generar un clasificador, el tamaño del gen que representa el algoritmo dependerá del número de algoritmos disponibles. Por otra parte, si se considera que los parámetros de aprendizaje de los algoritmos deben formar parte de la tarea de optimización, se usan, además del gen que representa el nombre del algoritmo, una serie de genes que representan los parámetros de aprendizaje de éste. El tamaño de cada uno de estos genes depende de los parámetros de aprendizaje que representen, por lo que el tamaño del cromosoma depende del número de genes que se usen para representar un algoritmo.

En la Figura 2.6 se muestra la codificación de un individuo en donde los primeros cuatros genes del cromosoma representan los cuatro algoritmos de aprendizaje a partir de los cuales se construirán los clasificadores de nivel-0 y el último gen representa el algoritmo a partir del cual se construirá el clasificador de nivel-1. La longitud de cada gen es de tres bits, razón por la cual se pueden seleccionar de entre 7 algoritmos de aprendizaje y la no presencia de ninguno.

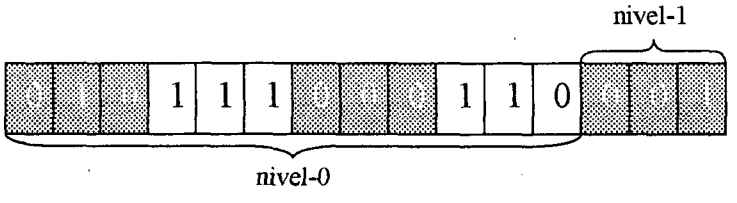


Figura 2.6: Descripción de la codificación de un individuo de GA-Stacking

2.5.3 Evaluación del *fitness*

La evaluación u obtención del *fitness* de los individuos que conforman la población se lleva a cabo en dos etapas (Figura 2.5, Fase II). La primera de estas (Figura 2.7 [a])se lleva a cabo mediante la decodificación del individuo y, basándose en esta representación, la selección de los algoritmos que serán usados para generar los clasificadores del conjunto. Cuando los clasificadores han sido seleccionados, se procede a la generación del conjunto de clasificadores, usando una parte del conjunto de datos disponibles (datos de entrenamiento). Al finalizar esta etapa, se tiene como resultado el conjunto de clasificadores.

La segunda etapa (Figura 2.7 [b]) en el proceso de evaluación del *fitness* de los individuos consiste en estimar la precisión del conjunto de clasificadores sobre los datos de validación. Estos son los datos restantes del conjunto, los que no han sido usados en la construcción. De esta forma, el *fitness* o adecuación de cada individuo es el porcentaje de aciertos que obtenga el conjunto de clasificadores sobre las instancias del conjunto de datos de validación.

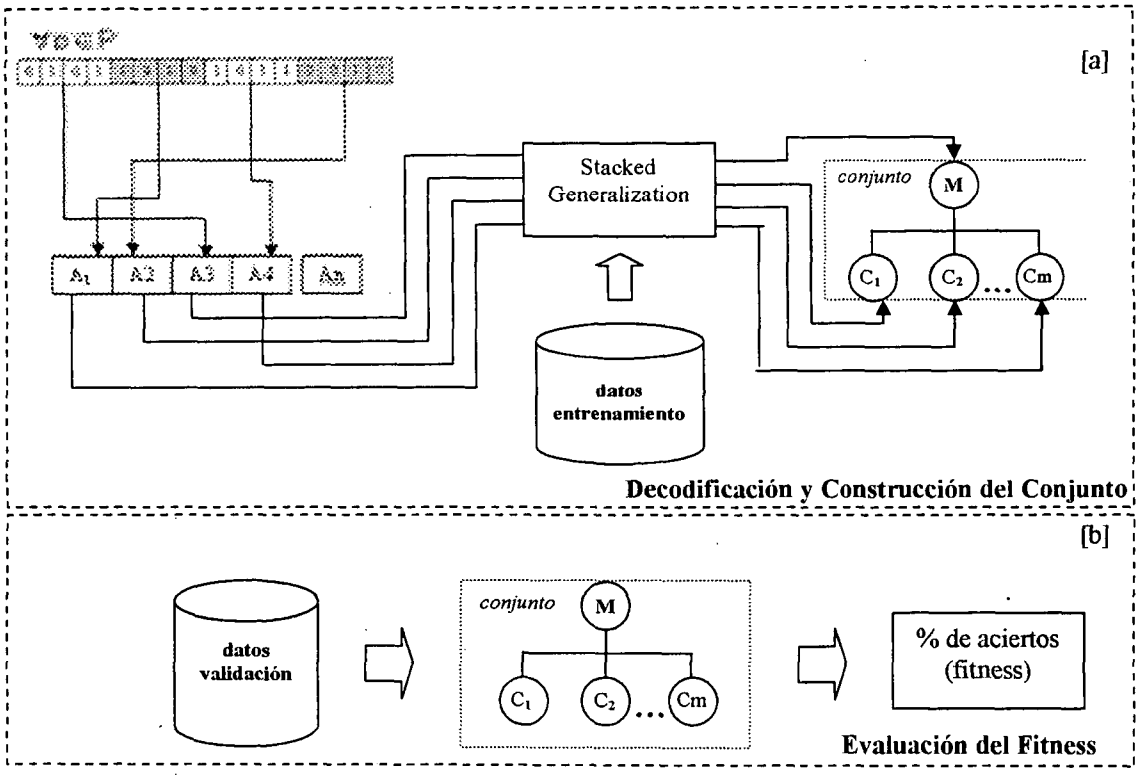


Figura 2.7: Evaluación del fitness en GA-Stacking

Una alternativa a dividir el conjunto de datos en entrenamiento y validación una única vez, es realizar una validación cruzada. En este caso el fitness será la media del porcentaje de aciertos de la validación cruzada, realizando así, una mejor estimación de la precisión de la solución.

2.6 Herramientas

En este apartado se da una breve introducción a las herramientas utilizadas para la realización de este Proyecto Fin de Carrera, además de resumir las características más importantes de cada una de ellas.

2.6.1 Weka

Weka [13] es el acrónimo de *Waikato Enviroment for Knowledge Analysis*. Weka es un paquete que implementa numerosos algoritmos de aprendizaje automático para tareas de minería de datos. Ha sido desarrollado usando el lenguaje Java por Ian Witten y Eibe Frank, ambos pertenecientes a la Universidad de Waikato en Nueva Zelanda.

Contiene diversas herramientas útiles para el preprocesado, clasificación, regresión, agrupamiento, y visualización de datos. Es también una herramienta adecuada para el desarrollo de nuevos esquemas de aprendizaje. Weka es un paquete *open-source* y se distribuye bajo la licencia *GNU Public License* (GPL).

Para proporcionar mayor flexibilidad, este paquete dispone de una API que permite a desarrolladores implementar nuevos algoritmos, así como modificar los algoritmos de aprendizaje que incluye.

2.6.2 Gajit

Gajit es el acrónimo de *Genetic Algorithm Java Implementation Toolkit*. Gajit es un paquete de propósito general que implementa la funcionalidad de un algoritmo genético, permitiendo al usuario definir y ejecutar sus propios problemas de

optimización. El paquete ha sido desarrollado usando el lenguaje de programación Java, y requiere únicamente de un compilador Java y conocimientos básicos de programación

Gajit ha sido diseñado, principalmente, para ser usado dentro de una aplicación y no como un *applet*, ya que requiere recompilar ciertas clases para poder ser ejecutado.

Gajit es un paquete *open-source* y se distribuye bajo la licencia *GNU Public License* versión 2 (GPL).

2.6.3 Eclipse

"Una especie de herramienta universal - un IDE abierto y extensible para todo y nada en particular."

Proyecto Eclipse

Eclipse es un entorno integrado de desarrollo (IDE) multiplataforma para crear aplicaciones clientes de cualquier tipo. Es libre y fue creado originalmente por IBM, aunque ahora lo desarrolla la Fundación Eclipse, una organización independiente sin ánimo de lucro que apoya una comunidad de código abierto. La aplicación más importante que ha sido realizada con este entorno es el afamado entorno de desarrollo integrado Java, llamado *Java Development Toolkit* (JDT). Este es el entorno de desarrollo que se está convirtiendo en el estándar de facto para Java, de hecho otros IDE's comerciales como *JBuilder* han anunciado que su próxima versión se basará en Eclipse.

Eclipse no es tan sólo un IDE, se trata de un *framework* ampliable mediante módulos (en inglés *plug-in*). Estos módulos o *plugins* proporcionan más funcionalidades, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Incluso hay *plugins* que permiten que el entorno de desarrollo soporte otros lenguajes además de Java, como por ejemplo PHP, Perl, etc.

Los componentes gráficos de Eclipse están basados en un juego de herramientas de tercera generación para Java de IBM llamado *SWT*, que mejora los de primera y

segunda generación de Sun. La interfaz de usuario de Eclipse cuenta con una capa intermedia de interfaz gráfica (GUI) llamada *JFace*, lo que simplifica la creación de aplicaciones basadas en *SWT*.

Capítulo 3

OBJETIVOS

En el capitulo anterior se ha presentado el estado de la cuestión en cuanto a conjuntos de clasificadores se refiere, así como la utilización de los AG's en el proceso de optimización de los mismos. Es este capítulo se presenta el objetivo general y los objetivos específicos de este Proyecto Fin de Carrera enmarcado en el área de conjuntos de clasificadores.

El objetivo general de este Proyecto Fin de Carrera es:

- Desarrollar un algoritmo optimizado de generación de conjuntos de clasificadores heterogéneos basándose en el algoritmo *GA-Stacking* [23].

Los objetivos específicos son:

- Determinar los métodos de generación de clasificadores más adecuados al proceso de creación del conjunto de clasificadores.

- Definir los parámetros adecuados de los algoritmos genéticos a utilizar para la generación del conjunto de clasificadores, haciendo especial hincapié en la

codificación del problema y en la función de evaluación de las soluciones generadas.

- Implementar el algoritmo desarrollado y todos los módulos necesarios, usando para ello las herramientas y recursos adecuados.

- Evaluar el algoritmo propuesto en distintos dominios de aplicación.

- Comparar la eficiencia del algoritmo desarrollado con *GA-Stacking*

Capítulo 4

GA-ENSEMBLE

En este capítulo se detalla el trabajo realizado durante este proyecto, partiendo de una descripción de alto nivel hasta los detalles.

En el primer apartado se describe el enfoque tomado por *GA-Ensemble*, que basándose en algoritmos genéticos busca obtener un conjunto de clasificadores optimizados para un problema dado. En este apartado se define el algoritmo, se presenta la codificación propuesta para la utilización de los AG's y se detalla el método de evaluación de las posibles soluciones encontradas por los AG's.

En el apartado 4.3 se presenta la implementación del algoritmo, incluyendo los módulos de los que consta, entradas y salidas de los mismos, etc. Dentro de este apartado también se muestra el modelo de conocimiento, en el cuál se comentan las clases de más alto nivel y sus atributos más significativos.

4.1 GA-Ensemble: Descripción del algoritmo

El término *GA-Ensemble* es el acrónico en inglés de *Genetic Algorithms for Ensambles of Classifiers*. *GA-Ensemble* trata la cuestión de definir, para un dominio dado, cuáles y cuántos algoritmos de aprendizaje son necesarios para formar el conjunto de clasificadores y cómo se combinan para resolver el problema de forma óptima. Esta búsqueda la aborda como un problema de optimización, el cual resuelve mediante la aplicación de algoritmos genéticos.

Uno de los objetivos de *GA-Ensemble* es aumentar la eficiencia de *GA-Stacking*, el algoritmo en el que se basa y que está definido en [23]. Para poder lograr este objetivo es necesario reducir el número de clasificadores que se necesita entrenar. Para evaluar cada individuo de la población del algoritmo genético *GA-Stacking* tiene que generar tantos clasificadores de nivel-0 como algoritmos representados se encuentren en el individuo. Además es necesario entrenar los meta-datos para entrenar el metaclasificador. Este es un proceso computacionalmente costoso. Para abordar este problema se ha decidido hacer uso de un *pool* de clasificadores entrenados, o lo que es lo mismo, entrenar todos los algoritmos que vayan a tomar parte en la ejecución del algoritmo genético *a priori* para evitar de ese modo tener que entrenarlos en sucesivas generaciones del algoritmo genético.

En la Figura 4.1 puede verse el esquema general propuesto, en donde se aprecia como, tomando como entrada los datos del dominio de aplicación, los algoritmos de aprendizaje generan el *pool* de clasificadores. Posteriormente se aplican los algoritmos genéticos con la finalidad de obtener como salida del sistema la configuración óptima del conjunto de clasificadores, para el dominio dado.

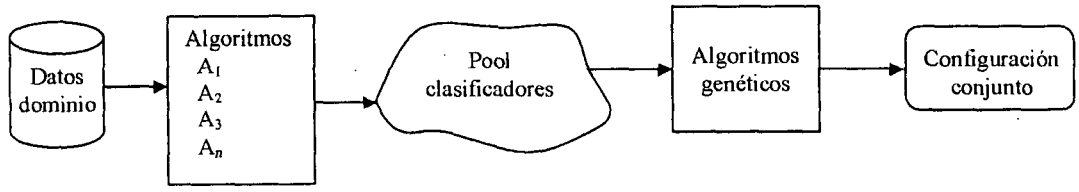


Figura 4.1: Esquema general GA-Ensemble

El uso de los algoritmos genéticos en un problema de optimización requiere, principalmente, definir dos características: la especificación de la codificación de las soluciones y la definición de la función de adecuación o *fitness*. *GA-Ensemble* tiene dos fases que pueden considerarse como fases previas a la ejecución de los algoritmos genéticos en sí. En la primera fase (Figura 4.2 [a]) se genera el *pool* de clasificadores con los datos del dominio de aplicación y los algoritmos de aprendizaje. El proceso de codificación de las soluciones se produce en la segunda fase (Figura 4.2 [b]). La evaluación del *fitness* es un proceso iterativo que se lleva a cabo en cada generación de los algoritmos genéticos (Figura 4.2 [c]) sobre todos los individuos de la población. A continuación se detalla la codificación de las soluciones utilizada y la evaluación de la función de *fitness*.

4.1.1 Codificación de las soluciones

Hay diversos modos en los que un problema puede ser codificado para ser resuelto mediante algoritmos genéticos (codificación binaria, hexadecimal, etc.). Para representar las soluciones según el enfoque que *GA-Ensemble* propone, se ha decidido hacer uso de una codificación binaria, puesto que ésta permite el empleo de AG's canónicos.

Se proponen dos formas de codificación de los individuos (soluciones) de acuerdo al método de combinación que se utilice. Según que tipo de codificación se use varía el tamaño del cromosoma. En ambas codificaciones el tamaño del cromosoma que representa al individuo está directamente definido por el número de clasificadores que forman parte del *pool* de clasificadores. Esto es así porque el hecho de que un clasificador forme parte o no de la solución, está representado de forma binaria por un

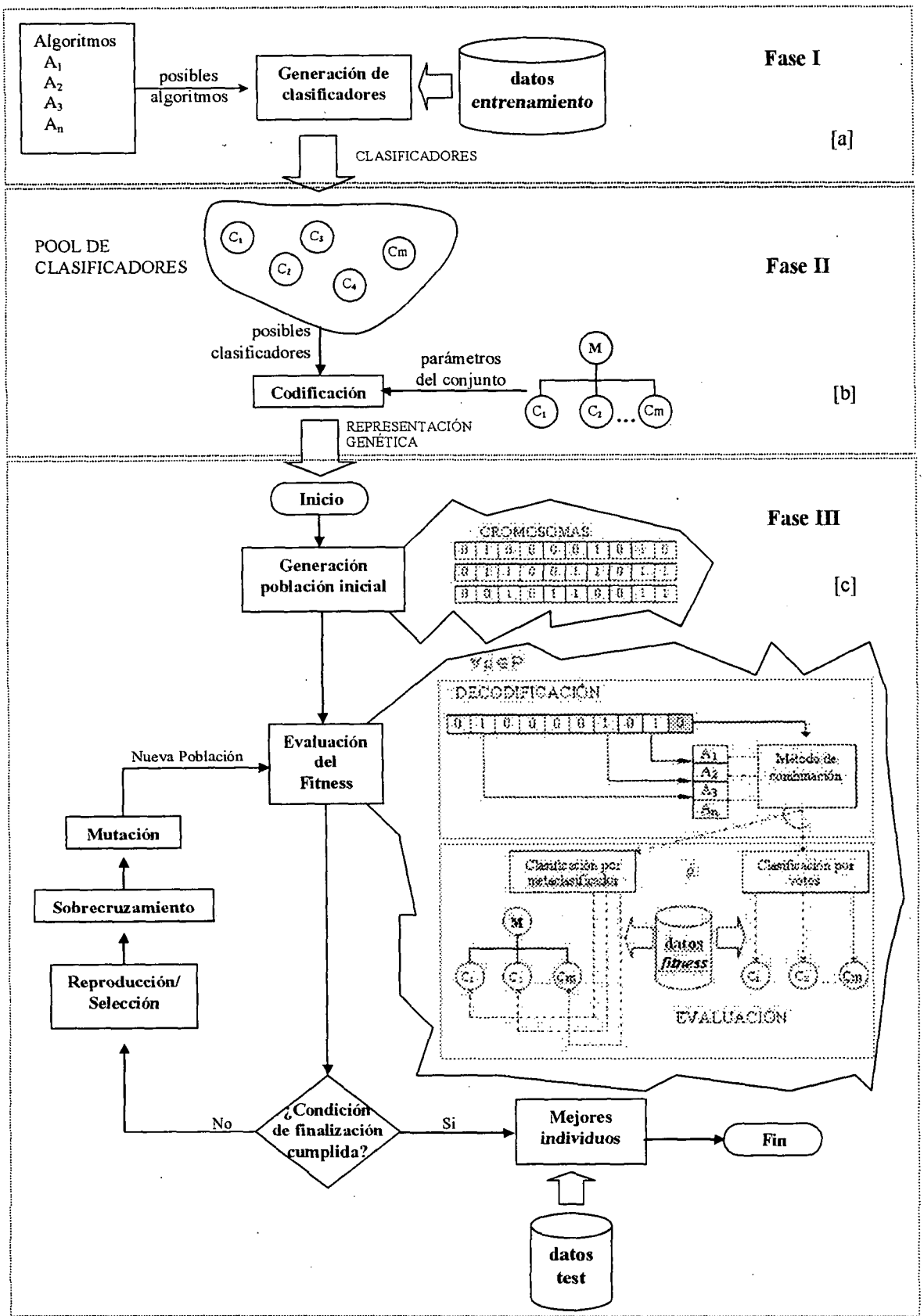


Figura 4.2: Marco propuesto: GA-Ensemble

gen del cromosoma. Según esta representación todos los clasificadores están representados en cada individuo, indicando si formarán parte del conjunto de clasificadores o no. Pero, como ya se ha mencionado, lo que realmente influye la utilización de una codificación u otra es el método para combinar los clasificadores.

En la primera codificación el algoritmo utiliza siempre un metaclasificador para combinar los clasificadores, por que lo que no es necesario añadir ninguna información al cromosoma y la longitud no varía. En otras palabras, existen tantos genes como número de clasificadores en el *pool*.

Por otra parte si se decide que el algoritmo pueda seleccionar, para combinar los clasificadores, entre usar un metaclasificador o combinarlos por votos, la codificación del cromosoma varía, utilizando además de los genes que representan los clasificadores, otro gen que se añade al final cromosoma. Este último gen del cromosoma es interpretado como un valor booleano, decidiendo cual de los métodos de combinación (por metaclasificador o por votos) se ha de emplear, para combinar los clasificadores que aparecen como miembros del conjunto.

Por ejemplo, si se decidiese que los clasificadores se combinen solo usando un metaclasificador y el *pool* de clasificadores lo formasen 10 clasificadores, la longitud del individuo sería de 10 genes de un bit, cada gen indica un clasificador. Sin embargo, si se optase por que el algoritmo haga también uso de la combinación por votos la longitud de un individuo sería entonces de 11 genes, los 10 primeros indican cuales de esos clasificadores constituyen el conjunto y el último indica que método usar para combinarlos.

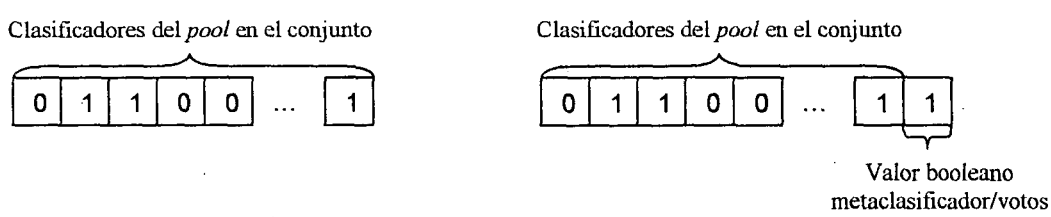


Figura 4.3: Descripción de la codificación binaria del individuo

En la Figura 4.3 se aprecia como la longitud en bits del cromosoma se deriva del número de clasificadores que puede formar parte del conjunto de clasificadores.

4.1.2 Evaluación del *fitness*

Para llevar a cabo el proceso de evaluación u obtención del *fitness* de los individuos que conforman la población se realiza los siguientes pasos. Primero se decodifica el individuo, y basándose en esta representación se seleccionan los clasificadores que formaran parte del conjunto de clasificadores. El siguiente paso es generar con los clasificadores el conjunto utilizando el método indicado (Figura 4.4 [a]), para lo cual se comprueba si se ha decidido utilizar solo metaclasificador como método de combinación o también se ha incluido la opción de combinación por votos, tal y como se detalla en la Figura 4.3. Nótese que en el caso de que se use un metaclasificador, bien porque sea la única opción o porque esté codificado en el cromosoma, será necesario entrenar el metaclasificador, pues los clasificadores que ya están entrenados son los clasificadores base.

Una vez el conjunto esta creado el siguiente paso es estimar la precisión del mismo sobre un conjunto de datos que no ha sido utilizado en la construcción del conjunto, es decir, que no ha sido utilizado para entrenar los clasificadores del *pool* (Figura 4.4 [b]). Este conjunto de datos es el llamado datos de *fitness*. La adecuación de cada individuo será el porcentaje de aciertos que obtenga el conjunto de clasificadores sobre las instancias del conjunto de datos de *fitness*.

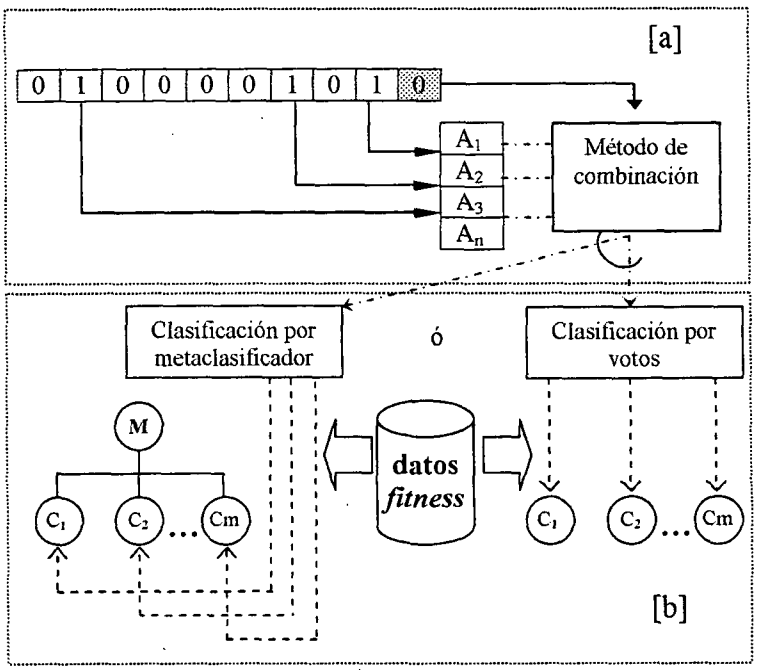


Figura 4.4: Evaluación del fitness en GA-Ensemble

El método que se ha utilizado para dividir el conjunto de datos en entrenamiento y *fitness* es mediante una validación cruzada. De este modo, el *fitness* es la media del porcentaje de aciertos de la validación cruzada, realizando así, una mejor estimación en la precisión de la solución.

4.2. Implementación de GA-Ensemble

4.2.1 Arquitectura de la aplicación

En este apartado se detalla la arquitectura que se ha usado para la implementación del algoritmo. Para ello se presentan los módulos de los que consta la aplicación, así como sus interfaces, detallando las entradas y salidas de los mismos y los datos que intercambian con otros módulos.

En al Figura 4.5 se detallan los módulos de los que consta la aplicación y la interacción entre ellos. Para una mayor claridad se detalla la interfaz de cada módulo de forma individual, incluyendo los datos que envía o recibe.

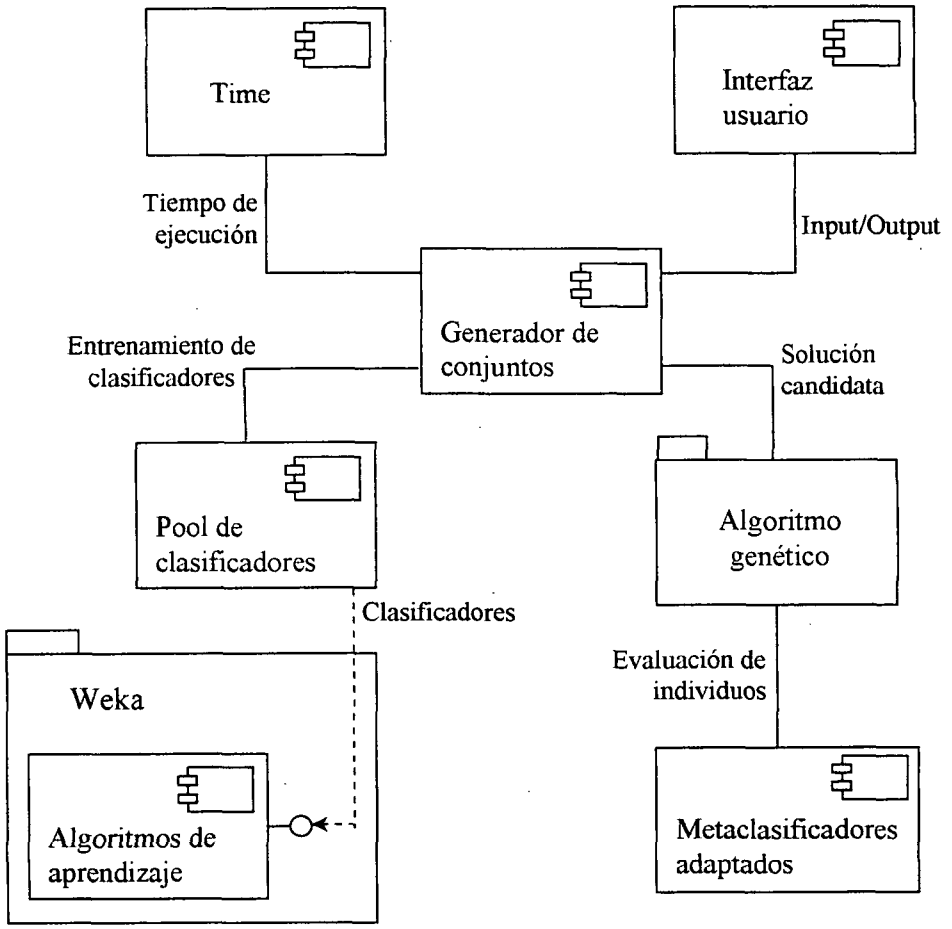


Figura 4.5: Módulos de la aplicación

A continuación se detalla la interacción que se lleva a cabo entre cada par de módulos:

- Interfaz: Time – Generador de conjuntos

  El reloj indica al generador de conjuntos el tiempo que ha llevado la ejecución de la aplicación.

- Interfaz: Generador de conjuntos – Interfaz usuario

  El generador de conjuntos recibe los parámetros de la aplicación en un fichero de texto y devuelve la salida de la aplicación por pantalla o en un fichero de texto.

- Interfaz: Pool de clasificadores – Generador de conjuntos

  El generador de conjuntos envía los datos del dominio e información de los parámetros al pool de clasificadores, y recibe los clasificadores entrenados.

- Interfaz: Pool de clasificadores – Algoritmos de aprendizaje

  El generador de conjuntos envía los datos del dominio relativos a los algoritmos de aprendizaje que necesita y los recibe estos algoritmos.

- Interfaz: Generador de conjuntos – Algoritmo genético

  El generador de conjuntos envía al algoritmo genético información relativa a los datos del dominio, junto a los clasificadores entrenados e información de los parámetros. Recibe de este la solución candidata, es decir, los clasificadores que formarán parte del conjunto de clasificadores.

- Interfaz: Algoritmo genético – Metaclasificadores adaptados

  El algoritmo genético envía información del dominio y los clasificadores a combinar a los metaclasificadores, estos devuelven la tasa de acierto del conjunto generado.

Para poder ilustrar de forma más clara como interactúan los diferentes módulos de la aplicación, se describe la interacción de estos como si de un algoritmo a muy alto nivel se tratase.

- El generador de conjuntos recibe los parámetros y la información del dominio del interfaz de usuario.
- El módulo time comienza a medir el tiempo.
- El generador de conjuntos manda parte de los parámetros y la información del dominio al pool de clasificadores.
- Según los datos recibidos, el pool de clasificadores solicita unos algoritmos u otros, y una vez los ha recibido los entrena creando el pool de clasificadores que envía al generador de conjuntos.
- El generador de conjuntos manda estos clasificadores ya entrenados junto a parte de los parámetros de entrada al algoritmo genético para que realice la búsqueda.
- Para la evaluación de cada individuo del algoritmo genético:
    - Enviar a los metaclasificadores adaptados los clasificadores que codifica el individuo y el método de combinación.
        - Los metaclasificadores crean el conjunto de clasificadores y calculan su precisión que envían al algoritmo genético
    - El algoritmo genético recibe la precisión para usarla como *fitness* del individuo
- El algoritmo genético envía la solución al generador de conjuntos, es decir, los clasificadores que al combinar han obtenido una mayor precisión
- El generador de conjuntos envía la solución y el tiempo de ejecución a la interfaz de usuario.

4.2.2 Descripción de los módulos

Este apartado servirá para dar una visión de alto nivel de los módulos de la aplicación (Figura 4.5). Para cada módulo se definirá su funcionalidad, entradas y salidas, así como los principales algoritmos.

Modulo *Time*

Este módulo es bastante simple. Recibe un evento que le indica que debe empezar a contabilizar el tiempo. Cuando recibe otro que le indica que debe parar, calcula la diferencia para calcular el tiempo transcurrido y lo envía por la salida estándar.

Módulo *Interfaz usuario*

Este módulo es el que se encarga de las entradas y salidas del sistema. Recibe los parámetros de la aplicación, los valida y los envía al módulo *Generador de conjuntos*. Recibe la información de salida de la aplicación y según su configuración, a través de los parámetros de entrada, la manda a un fichero o la muestra por pantalla.

Módulo *Algoritmos de aprendizaje*

Este módulo es el que contiene todos los algoritmos de aprendizaje que se usarán para generar los clasificadores base del conjunto de clasificadores. En él están todas las implementaciones de los posibles algoritmos. Cada algoritmo esta implementando para recibir únicamente los datos de entrenamiento y mediante un método generar el clasificador, devuelve el clasificador solicitado como un objeto de tipo *Classifier*, definida en el API de Weka.

Módulo *Pool de clasificadores*

Este módulo se encarga de la creación de los *pools* de clasificadores necesarios. Recibe los datos de entrenamiento y los algoritmos de aprendizaje que utilizará para construir el *pool* de clasificadores. Toma uno a uno todos los clasificadores que deben formar parte del *pool*. Para cada uno, haciendo uso del módulo *Algoritmos de aprendizaje*, crea una instancia del algoritmo correspondiente y lo entrena utilizando los datos de entrenamiento. Una vez el clasificador está entrenado lo añade al *pool*. Cuando el *pool* esta completo lo devuelve al módulo *Generador de conjuntos* junto al conjunto de datos que sirvió para entrenarlo. La evolución del proceso de entrenamiento de cada algoritmo y su posterior agregación al pool se va registrando en un fichero de *log*, para poder realizar un seguimiento en caso de que surjan problemas.

Módulo *Algoritmo genético*

El módulo *Algoritmo genético* engloba toda la funcionalidad relativa al algoritmo evolutivo, desde la codificación de las soluciones hasta la evaluación de la función de *fitness*. Recibe del módulo *Generador de conjuntos* los parámetros de

entrada relativos al AG así como toda la información de los *pools* de clasificadores, esto es, los clasificadores ya entrenados, el conjunto de datos que se ha utilizado para entrenarlos y el conjunto de datos que se debe usar para validarlos. Este módulo se encarga de generar la población inicial del AG, basándose en los parámetros recibidos, para después iniciar la ejecución del proceso. Para poder calcular el *fitness* de cada individuo utiliza el *pool* de clasificadores, calculando la precisión del conjunto de clasificadores generado con los clasificadores que codifica el cromosoma. En el proceso de evaluar a un individuo se debe generar un conjunto de clasificadores, y para esto se utiliza del módulo *Metaclasificadores adaptados*. Una vez termina la ejecución del AG, devuelve al módulo *Generador de conjuntos* un vector en el que está almacenado el mejor individuo de cada generación.

Módulo *Metaclasificadores adaptados*

Este módulo es el encargado de combinar los diversos clasificadores. Recibe del módulo *Algoritmo genético* los clasificadores, previamente entrenados, que se deben combinar, las instancias usadas para entrenar a los clasificadores del pool y las instancias con las que se debe evaluar el conjunto que se genere (los datos de *fitness*). Genera el conjunto de clasificadores solicitado por *Algoritmo genético,* se evalúa usando los datos facilitados y devuelve su éxito en forma de porcentaje de acierto. Este porcentaje de acierto es lo que usará el AG como *fitness* de los individuos.

Módulo *Generador de conjuntos*

Este módulo es el principal de la aplicación y es el que contiene la funcionalidad relativa a *GA-Ensemble*. Esta funcionalidad se puede describir mediante el pseudocódigo mostrado en la Figura 4.6.

Pseudocódigo: Módulo Generador de conjuntos

- Recibir de Interfaz usuario los parámetros de la aplicación
- Obtener los datos del dominio
- Enviar evento al módulo Time para iniciar la medición del tiempo.
- Comenzar el proceso de validación cruzada

  Mientras queden iteraciones de la validación cruzada
  - Dividir los datos del dominio en datos de entrenamiento y datos de test
  - Dividir los datos de entrenamiento en datos de entrenamiento y datos de *fitness*
  - Obtener los algoritmos de aprendizaje
  - Enviar los algoritmos de aprendizaje junto a los datos de entrenamiento al módulo *Pool de Clasificadores.*
  - Recibir los clasificadores entrenados en el *pool*
  - Enviar los clasificadores entrenados, los datos de entrenamiento, los datos de *fitness* e información de los parámetros al módulo *Algoritmo genético.*
  - Recibir un vector con los mejores individuos.
  - Generar los conjuntos de clasificadores que definen las soluciones y validarlos con los datos de test.
  - Almacenar su precisión.

  Fin-mientras
- Promediar el porcentaje de éxito obtenido por la soluciones en cada iteración de la validación cruzada.
- Enviar evento al módulo Time para finalizar el cronometraje.
- Enviar datos de salida al usuario a través del modulo Interfaz Usuario

Figura 4.6: Pseudocódigo del módulo Generador de conjuntos

4.2.3 Modelo de conocimiento de la aplicación

En este apartado se comentan a un alto nivel las clases más relevantes así como sus atributos más significativos. Se debe reseñar que, normalmente, la funcionalidad de un clasificador se define en una sola clase, siendo esta ocasión un caso un poco especial, ya que debido al tamaño y complejidad del algoritmo se han necesitado diversas clases. Cada una de las clases consideradas principales se describe a continuación de forma breve con sus atributos más importantes.

Clase *Ensemble*: Esta clase es la que define la propia funcionalidad del algoritmo de optimización de conjuntos de clasificadores, y por eso la principal. Esta clase contiene el código ejecutable del algoritmo y es por lo tanto la que se encarga de crear el *pool* de clasificadores o configurar y ejecutar el algoritmo genético, entre otras cosas.

Clase *Clock*: Esta clase es muy sencilla pero importante, pues gracias a esta clase se calcula el tiempo de ejecución de *GA-Ensemble*. Tan solo almacena una hora de inicio y una de fin.

Clase *ClassifiersPool*: Esta es una clase vital para *GA-Ensemble* pues esta clase contiene la funcionalidad necesaria para construir el *pool* de clasificadores. Sus atributos más importantes son:

- *m_trainingdata:* Almacena los datos que se usan para entrenar a los clasificadores del *pool*.
- *classifiersVector*: Atributo tipo vector, que almacena el nombre de todos los clasificadores que formarán parte del *pool*.

Clase *Genetico*: La importancia de esta clase radica en que es la que contiene la funcionalidad del algoritmo genético. Tiene numerosos atributos, la mayoría relacionamos con los parámetros del AG, tales como el tamaño de la población o el número de generaciones. Pero también tiene otros atributos, entre ellos los más relevantes son:

- *m_PoolsDataVector:* Atributo tipo vector, almacena los datos relativos a los *pools* de clasificadores que va a necesitar usar el AG. Contiene información como el conjunto de datos usado para el entrenamiento o los propios clasificadores entrenados.
- *AgMeta*: Atributo tipo booleano, indica al AG de que manera se deben combinar los clasificadores (únicamente por metaclasificador o también usando votos) para hallar el *fitness* de cada individuo.

Clase My_Stacking: Esta es una de las clases base del algoritmo, ya que es la que implementa el método de combinación por metaclasificador para clasificadores base ya entrenados. Esta basado en el algoritmo *Stacking* implementado en Weka, de ahí su nombre. Es necesario incluirle previamente a su construcción el conjunto de clasificadores base y es obligatorio que estos se encuentren ya entrenados. Uno de sus atributos más importante es:

- *m_MetaClassifier:* Atributo tipo *Classifier*, define cual será el algoritmo a emplear como metaclasificador.

Clase My_Bagging: Esta clase es similar a la anterior, ya que es la que implementa el otro método de combinación de clasificadores que permite *GA-Ensemble*, es decir, el método de combinación por votos. Esta basado en el algoritmo *Bagging* implementado en Weka. Para poder generar el conjunto, necesita disponer previamente de los clasificadores base, y que estos se encuentren ya entrenados.

Capítulo 5

EVALUACIÓN EXPERIMENTAL

Hasta este punto se ha dado una visión general del funcionamiento teórico del algoritmo de optimización de conjuntos de clasificadores desarrollado en este Proyecto Fin de Carrera. Es en este capítulo donde se detalla el proceso de evaluación del algoritmo desarrollado. Se presentan los resultados, los parámetros utilizados y, en general, información relevante respecto a los resultados de la experimentación.

En el primer apartado se describen los conjuntos de datos utilizados en la experimentación. En el apartado 5.2 se detallan los algoritmos de aprendizaje utilizados por *GA-Ensemble*. En el tercer apartado se enumeran todos los parámetros que se han utilizado durante las pruebas. El apartado 5.4 sirve para mostrar los resultados preliminares del algoritmo. Por último, en el apartado 5.5, se incluyen los resultados finales obtenidos, así como su valoración

5.1 Dominios

Para llevar a cabo los experimentos se han utilizado diversos dominios del conocido repositorio de datos del UCI [1]. Estos dominios han sido ampliamente utilizados en otros estudios. Los dominios seleccionados se muestran en la tabla 5.1.

Dominio	Atributos	Instancias	Clases
australian	14	690	2
car	6	1728	4
chess	36	3196	2
diabetes	8	768	2
glass	9	214	6
hepatitis	19	155	2
hypo	25	3163	2

Tabla 5.1: Dominios utilizados en la experimentación

5.2 Algoritmos de aprendizaje

Para poder obtener la combinación óptima de clasificadores se ha utilizado un grupo de 15 algoritmos de aprendizaje para generar los clasificadores base o de nivel-0. Este conjunto ha sido considerado como el más adecuado para el propósito de este Proyecto Fin de Carrera, basándose en los resultados descritos en [23]. Los algoritmos utilizados para generar los miembros del conjunto se indican a continuación:

- ✓ Clasificador probabilístico NaiveBayes [21].
- ✓ PART [14]. Clasificador basado en reglas, forma una lista de árboles de decisión parcialmente podados usando la heurística de C4.5.
- ✓ J48 [30]. Genera árboles de decisión.
- ✓ J48 sin poda (opción -U).
- ✓ DecisionStump [20]. Genera árboles de decisión de un solo nivel.
- ✓ DecisionTable [22]. Es un clasificador simple que utiliza la clase mayoritaria.
- ✓ Clasificador basado en reglas DecisionTable usando el vecino más cercano (opción -I)
- ✓ ClassificationViaRegression [4]. Clasifica haciendo uso de métodos de regresión.
- ✓ RandomForest. Basado en árboles de decisión, combina un gran número de árboles de decisión sin podar.
- ✓ RandomTree. Basado en árboles de decisión, construye un árbol que considera K características aleatorias en cada nodo.

- ✓ Clasificador VFI [8].
- ✓ ConjunctiveRule. Basado en reglas conjuntivas simples que puede predecir usando clases nominales y numéricas.
- ✓ JRip [7]. Basado en reglas proposicionales.
- ✓ NNge [24]. Basado en reglas.
- ✓ HyperPipes [36]. Para cada categoría se construye un *HyperPipe* que contiene todos los puntos de esa categoría.

5.3 Parámetros de *GA-Ensemble*

En todos los experimentos realizados existen unos parámetros de configuración denominados generales, en otras palabras, estos parámetros se han mantenido constantes a lo largo de toda la experimentación. Estos parámetros corresponden, principalmente, al AG, sin embargo, como se verá a continuación, existen otros que son propios al algoritmo.

5.3.1 Parámetros propios del algoritmo genético

Los parámetros del algoritmo genético que no se han visto modificados a lo largo de las pruebas son:

- ✓ *Número de ejecuciones del algoritmo genético* – El valor de este parámetro ha sido siempre de tres, esto es, la ejecución del algoritmo genético se ha realizado siempre, tres veces, obteniendo como resultado final el mejor individuo de las tres ejecuciones.
- ✓ *Tasa de mutación* - La tasa de mutación usada ha sido siempre del 10%.
- ✓ *Tasa de elite* - Siempre se ha usado la misma tasa de elite, esta ha sido del 10%.
- ✓ *Tasa de desecho* - Al igual que la tasa de elite y de mutación, la tasa de desecho ha sido siempre constante, su valor ha sido de 5%.

5.3.2 Otros parámetros.

Existen otros parámetros importantes de *GA-Ensemble* que no se han modificado en ninguna de las pruebas:

- ✓ *Número de folders de la validación cruzada* – El número de *folders* que se ha usado para la validación cruzada y en los que, por tanto, se ha visto dividido el conjunto de datos inicial ha sido de diez *folders*. Se recuerda que según esta definido el algoritmo, en cada iteración uno de esos *folders* se usa para test y el resto para el entrenamiento.

- ✓ *Algoritmo de aprendizaje para generar el metaclasificador.* En todas las pruebas, en el caso de que los clasificadores debieran combinarse usando un metaclasificador, se ha utilizado el algoritmo *Classification-ViaRegression* [4]. Esto es así porque este clasificador es utilizado como metaclasificador por defecto en el algoritmo *Stacking*.

5.4 Resultados preliminares

Los primeros pasos dados en la experimentación fueron enfocados no a optimizar del algoritmo, siendo este uno de los objetivos de este trabajo, sino a comprobar que la solución aportada era valida y ofrecía unos resultados positivos. En este punto se busca ver como a medida que evolucionan los individuos de la población, la combinación de conjuntos de clasificadores obtenidos son más eficientes y obtienen mejores resultados en cada nueva generación.

Tal y como se ha explicado en el apartado 4.2.2., el conjunto de entrenamiento se divide a su vez en datos de entrenamiento y datos de *fitness*. La obtención de los diferentes conjuntos de datos se realizaba, durante las pruebas preliminares, dividendo simplemente el conjunto de datos de entrenamiento por la mitad, tal y como se muestra en la Figura 5.1. Con una parte se realizaba el entrenamiento de los algoritmos de aprendizaje, mientras que con la otra se realizaba el cálculo de la función de *fitness* para cada individuo.

Como se verá a continuación, la opción de utilzar una parte de los datos para el entrenamiento y otra para el *fitness* no ofreció los mejores resultados, lo que llevó a replantear el algoritmo desarrollado.

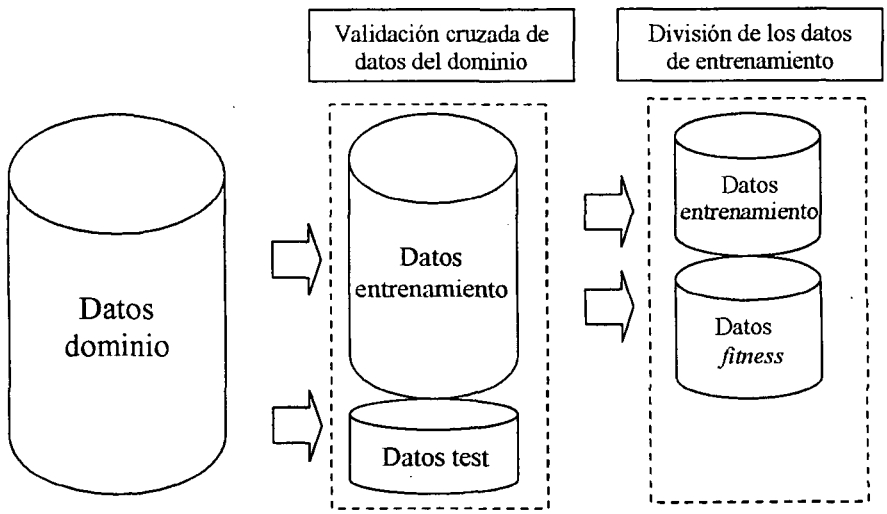


Figura 5.1: Conjuntos de datos preliminares de GA-Ensemble

5.4.1 Parámetros de las pruebas preliminares

Los parámetros específicos que se utilizaron durante las pruebas preliminares fueron los siguientes:

- ✓ *Número de generaciones del algoritmo genético* - Para estas primeras pruebas se utilizaron 50 generaciones.
- ✓ *Tamaño de la población* - La población del algoritmo genético constaba de 25 individuos.
- ✓ *Método de combinación de los clasificadores* – Según se ha explicado en el apartado 4.2.1, para estas pruebas se usó solo el primer tipo de codificación, o lo que es lo mismo, las clasificadores se combinaron, únicamente, haciendo uso de un metaclasificador.

5.4.2 Experimentación preliminar

Realizando las pruebas con los parámetros anteriormente indicados sobre tres conjuntos de datos diferentes se pudo observar como el *fitness* del algoritmo genético a partir de un punto se mostraba invariable, en otras palabras, no continuaba mejorando independientemente del conjunto de datos de entrada. Se comprobó que se daba la misma situación en todas las pruebas, en donde a partir de generaciones iniciales, el

fitness se estancaba. Después de analizar el desarrollo de la prueba se llegó a la conclusión que el algoritmo se sobreadaptaba a los datos usados para entrenamiento (Figura 5.1).

En la Figura 5.2 se puede apreciar el grado de sobreadaptación obtenida en las pruebas preliminares sobre los dominios utilizados. (*Hepatitis*, *Glass* y *Diabetes*).

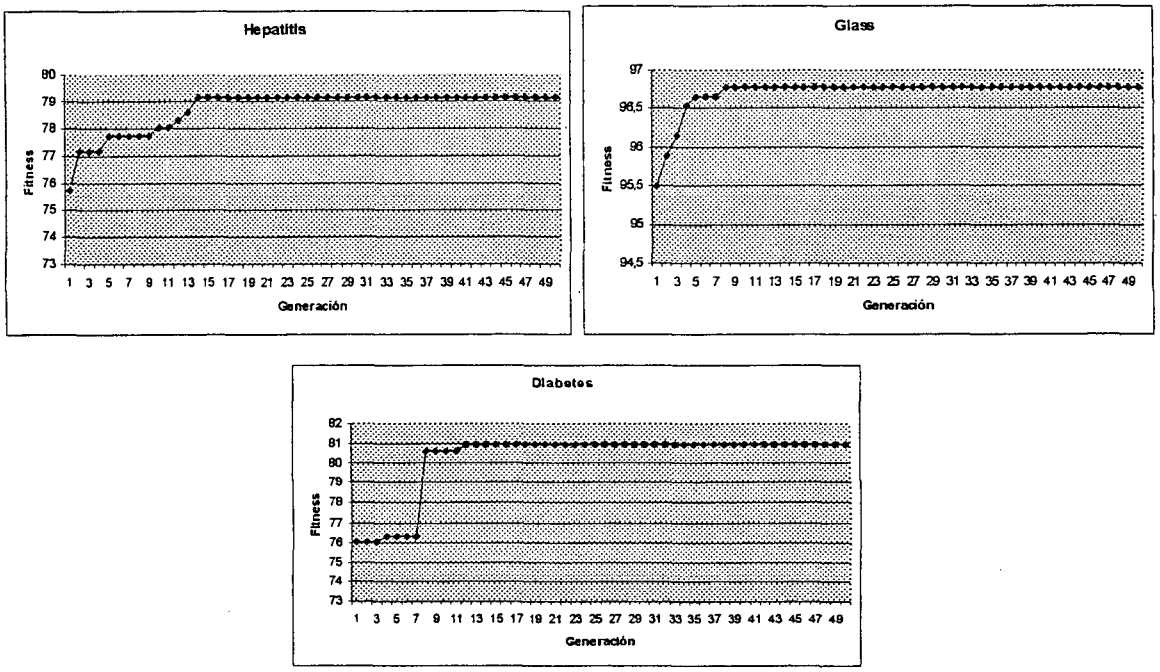


Figura 5.2: Pruebas preliminares de GA-Ensemble

Como se puede observar en las gráficas de resultados (Figura 5.2), utilizando la primera versión del algoritmo de optimización de conjuntos de clasificadores, la población del algoritmo genético evoluciona, únicamente, durante las primeras generaciones, quedándose estancada poco después, se observa como en el mejor de los casos el AG consigue evolucionar únicamente hasta la generación número 15, no cambiando el valor del *fitness* durante el resto de la ejecución.

También se comprobó como variando los parámetros del algoritmo genético no se obtenían mejoras en los resultados. Los cambios principalmente se enfocaron a aumentar la tasa de mutación de los individuos y el tamaño de la población, aumentando la tasa hasta llegar el 5% y el tamaño de la población hasta llegar a los 100 individuos, pero no se apreció ningún cambio significativo. Es a raíz de estos resultados cuando se constata que se debe realizar modificaciones en el algoritmo de optimización de conjuntos.

5.4.3 Evitando la sobreadaptación

En las pruebas preliminares se obtienen resultados mediocres debido a que el número de instancias del conjunto de datos del dominio que son utilizadas para construir el conjunto de clasificadores es muy bajo.

Para poder solucionar este problema el algoritmo de optimización de conjuntos se modificó. Se decidió que el método para dividir el conjunto de entrenamiento en datos de entrenamiento y datos de *fitness* fuese mediante una validación cruzada, siendo el *fitness* de cada individuo la media del porcentaje de aciertos de la validación cruzada.

Como se ve en al Figura 5.1, en las pruebas preliminares se dividía la mitad del conjunto de datos de entrenamiento en dos partes, utilizándose una mitad para el entrenamiento de los algoritmos de aprendizaje y la otra para el cálculo del *fitness* de los individuos de la población. Para evitar la sobreadaptación se decidió que tanto el entrenamiento de los algoritmos como el cálculo del *fitness* se llevase cabo con todo el conjunto de datos disponible para entrenamiento (Figura 5.3), mediante una validación cruzada.

Para poder comprobar si estos cambios suponían una mejora, se usó una validación cruzada de dos *folders*. En otras palabras, en esta ocasión la evaluación de cada individuo consta de los siguientes pasos:

1) Se divide el conjunto de datos de entrenamiento en dos subconjuntos de igual tamaño (Figura 5.3 [b]).

2) Primero se entrenan los algoritmos de aprendizaje con el primer subconjunto usando el segundo subconjunto para calcular el primer *fitness* parcial.
3) Después se entrenan los algoritmos de aprendizaje con el segundo subconjunto usando el primer subconjunto para calcular el segundo *fitness* parcial.
4) El *fitness* final del individuo es el promedio de los dos *fitness* parciales

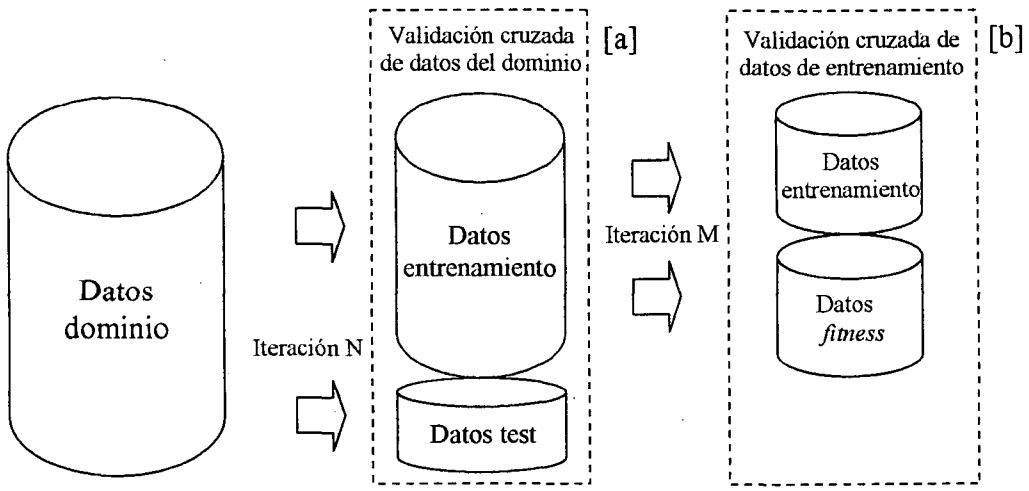


Figura 5.3: División de los datos en GA-Ensemble

De este modo todo el conjunto de datos es usado para el aprendizaje de los algoritmos de aprendizaje y para la evaluación del *fitness* del algoritmo genético, evitando así, teóricamente, los problemas de las pruebas anteriores.

Con este algoritmo se realizaron nuevamente las pruebas sobre los mismos conjuntos de datos obteniendo en este caso resultados diferentes, pues no se apreciaba que el algoritmos se sobreadaptase al conjunto de datos en modo alguno. En la Figura 5.4 se observa la evolución del *fitness* y el porcentaje de acierto sobre el conjunto de test de los AG's sobre el dominio de *Diabetes*.

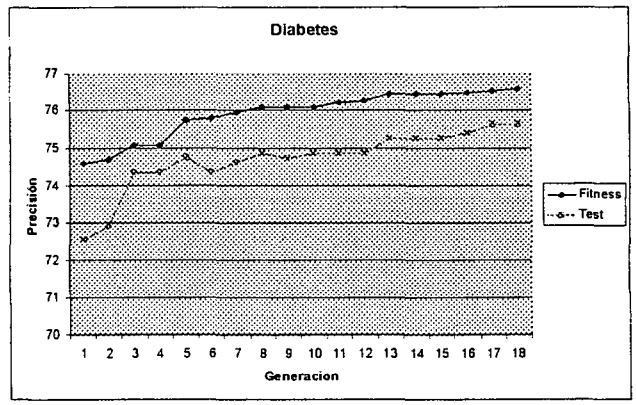


Figura 5.4: Diabetes con una validación cruzada de 2 para el conjunto de entrenamiento

5.5 Evaluación complementaria

Una vez definido el modo de dividir el conjunto de datos de entrenamiento se comienza el proceso de optimización del algoritmo. Se experimenta variando el número de *folders* de la validación cruzada del conjunto de entrenamiento, para comprobar en que medida estos cambios afectan a la precisión del algoritmo.

En teoría, dividir el conjunto de entrenamiento en un mayor número de segmentos, hace que el aprendizaje sea más completo, pero a su vez, realizar este proceso provoca que se reduzca el número de instancias dedicadas al *fitness*, por lo que es necesario mantener un equilibrio en esta división. En esta serie de experimentos se busca encontrar el número adecuado de *folders* en los que dividir el conjunto de entrenamiento. Es por esto que se añade un nuevo parámetro de entrada al algoritmo, el número de *folders* o divisiones en las que se va a segmentar el conjunto de datos de entrenamiento da cada iteración de la validación cruzada.

Otro punto a considerar en este bloque de experimentos es el tipo de codificación que se utiliza, la que permite combinar los clasificadores únicamente por metaclasificador o la que también permite combinarlos por votos. Una ventaja de permitir que el algoritmo pueda seleccionar el método de combinación entre metaclasificador o votos, es que el espacio de búsqueda aumenta, y es el propio

algoritmo genético el que determina cual es el método más adecuado a utilizar en cada ocasión.

5.5.1 Parámetros experimentales

Para la realización de estos experimentos se utilizan los siguientes valores de los parámetros del algoritmo:

- ✓ *Generaciones del algoritmo genético* - 30 generaciones.
- ✓ *Tamaño de la población* - La población del algoritmo genético constaba de 50 individuos.

De entre los parámetros que se usaron destacan por su importancia e influencia sobre los resultados del algoritmo:

- ✓ *Método de combinación de los clasificadores* – Los experimentos se llevaron a cabo usando las dos opciones posibles para combinar los clasificadores, por metaclasificador o por metaclasificador y votos.
- ✓ *División del conjunto de entrenamiento.* Se usaron diferente número de iteraciones para la validación cruzada del conjunto de entrenamiento (y por lo tanto variando también el número de partes en las que se dividían las instancias), se realizaron pruebas usando 3 iteraciones y pruebas usando 5 iteraciones

5.5.2 Experimentación

Las pruebas se realizaron, mezclando el método de combinación y el número de iteraciones para la validación cruzada del conjunto de entrenamiento, sobre todos los dominios definidos en el apartado 5.1.

En la Figuras 5.5, 5.6, 5.7, 5.8, 5.9, 5.10 y 5.11 se presenta de manera gráfica, el porcentaje medio de acierto sobre los datos de *fitness* y sobre los datos de test en cada generación del algoritmo genético. Este porcentaje es calculado como el promedio de los porcentajes de acierto obtenidos en cada una de las iteraciones de la validación cruzada que se realiza sobre los datos del dominio. Se comprueba como el valor del

fitness aumenta a medida que avanza el AG y como, a su vez, varía el acierto sobre los datos de test.

Dominio *Australian*

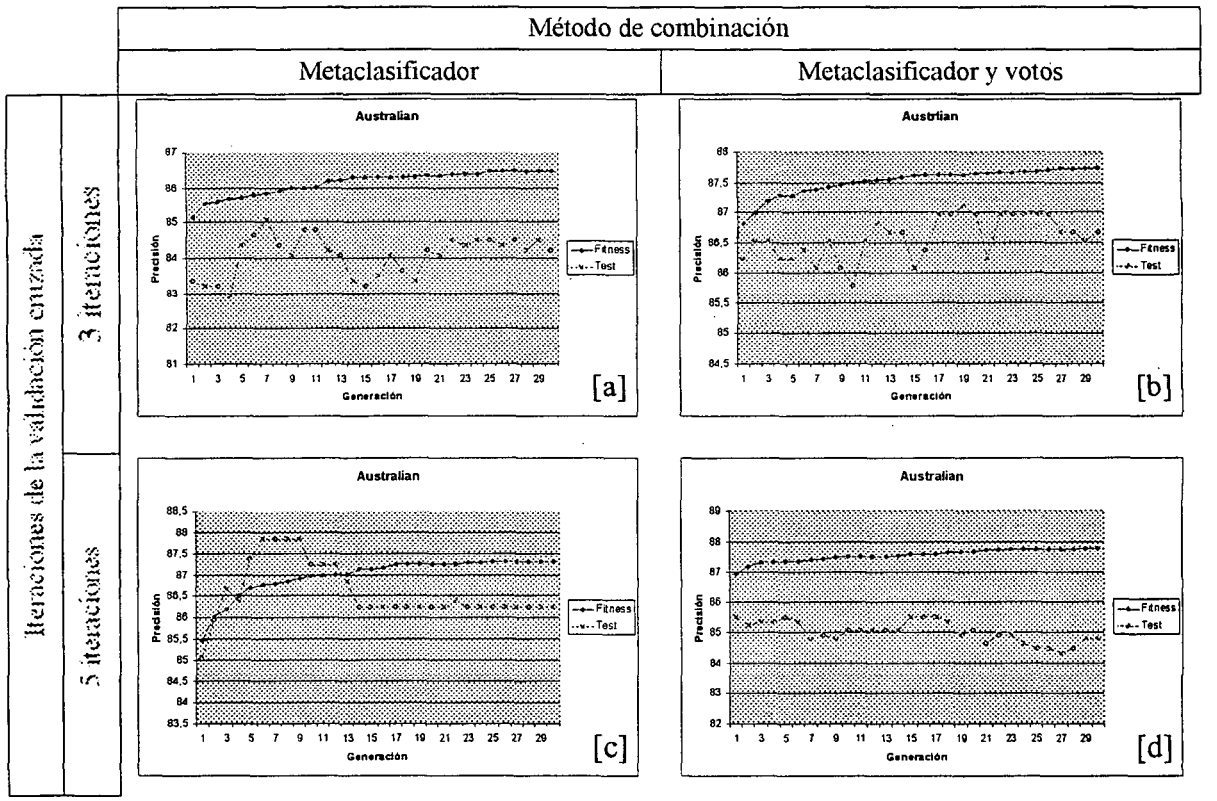


Figura 5.5: Resultados para el dominio Australian

En la Figura 5.5 se observa como los mejores resultados para el dominio *Australian* se obtienen en el caso de permitir combinación por metaclasificador y combinación por votos cuando el conjunto de entrenamiento se divide en 3 *folders* (Figura 5.5 [b]). La diferencia con el peor resultado obtenido (Figura 5.5 [a]), sin ser especialmente importante, es reseñable puesto que esta diferencia es del 2,46%. Otro dato a considerar es que en el caso de usar como método de combinación únicamente metaclasificador y dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.5 [c]) el porcentaje de acierto obtenido es también muy bueno, existiendo tan solo una diferencia del 0,03% respecto al mejor.

Dominio *Diabetes*

Sobre este dominio, además de las pruebas usando 3 y 5 iteraciones para la validación cruzada del conjunto de datos de entrenamiento, también se realizaron pruebas usando 7 iteraciones.

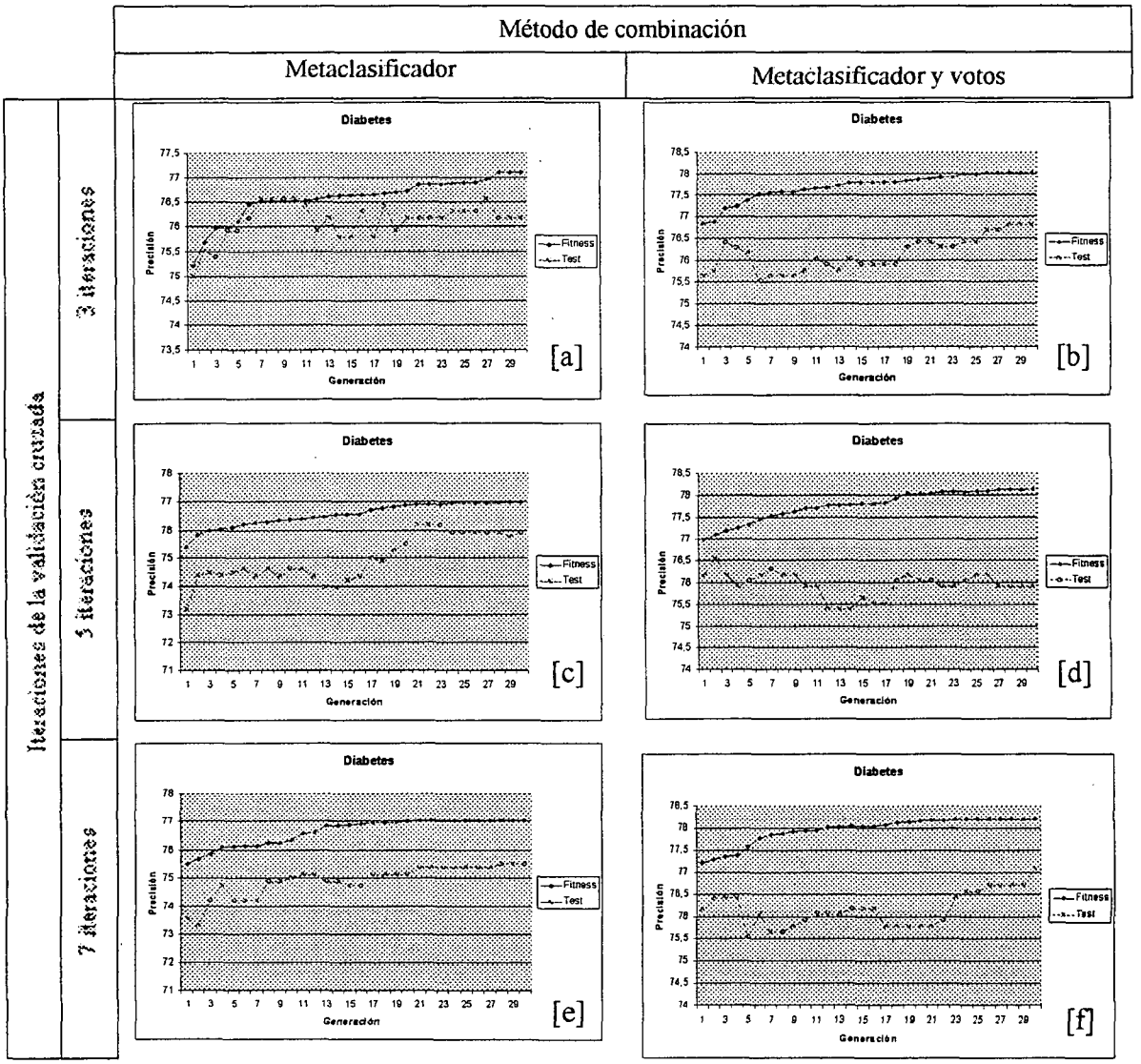


Figura 5.6: Resultados para el dominio Diabetes

Como se observa en la Figura 5.6 los mejores resultados para el dominio *Diabetes* se obtienen en el caso de permitir combinación por metaclasificador y combinación por votos cuando el conjunto de entrenamiento se divide en 7 *folders* (Figura 5.6 [f]), en dicho caso el porcentaje acierto es del 77,08%. En este bloque de pruebas, los resultados obtenidos tampoco son demasiado diferentes entre ellos, pues el peor resultado obtenido es en el caso de usar combinación por metaclasificador y dividir el conjunto de datos en 7 *folders* (Figura 5.6 [e]), obteniéndose una precisión del 75,51%, valor que no puede considerarse mediocre.

Dominio *Car*

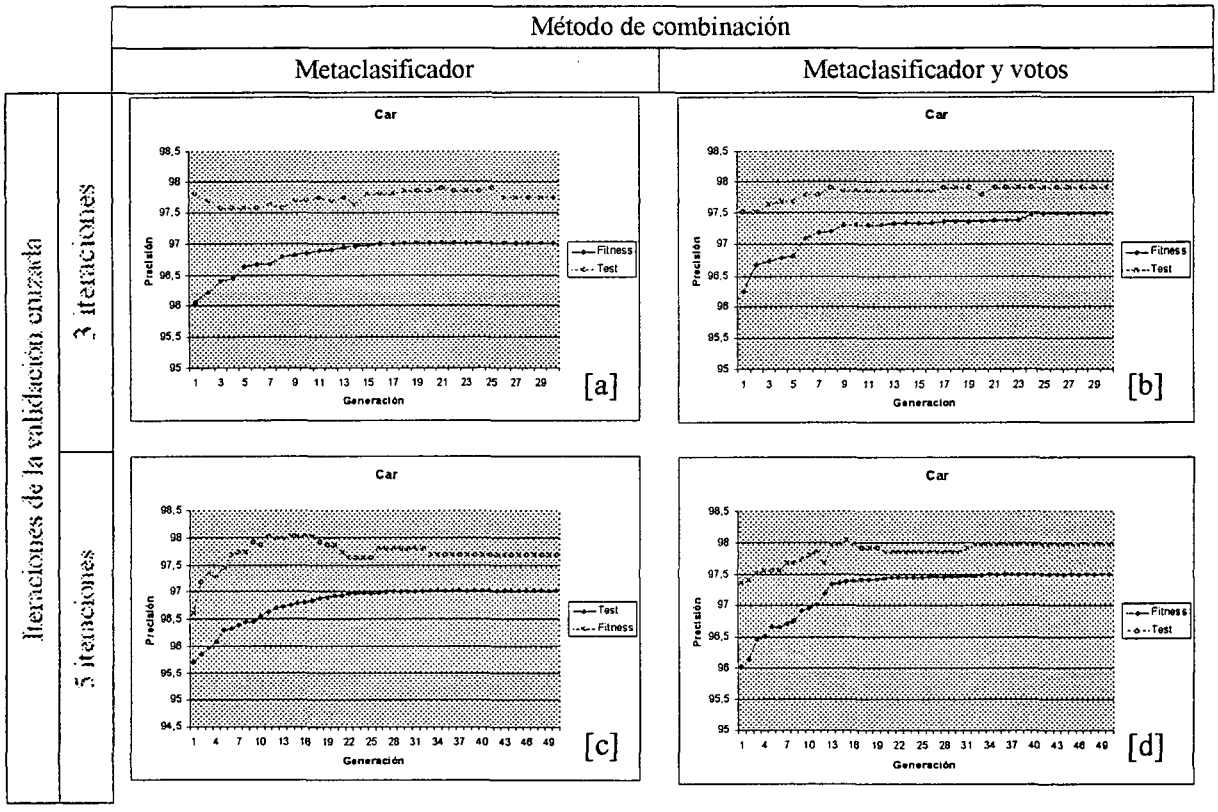


Figura 5.7: Resultados para el dominio Car

Al examinar la Figura 5.7 se observa como los resultados para el dominio *Cars* tienen unos valores situados entre el 97,68% y el 97,97%, existiendo una diferencia de 0,02% entre el mejor y el peor resultado. El mejor resultado se obtiene en el caso de permitir combinación por metaclasificador y combinación por votos cuando el conjunto de entrenamiento se divide en 5 *folders* (Figura 5.7 [d]), aunque no es significativamente superior al resto. El que caso de usar únicamente combinación por metaclasificador, se obtiene una precisión de 97,64% al dividir el conjunto de entrenamiento en 3 *folders* (Figura 5.7 [a]) y de 97,91% al dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.7 [c]).

Dominio *Chess*

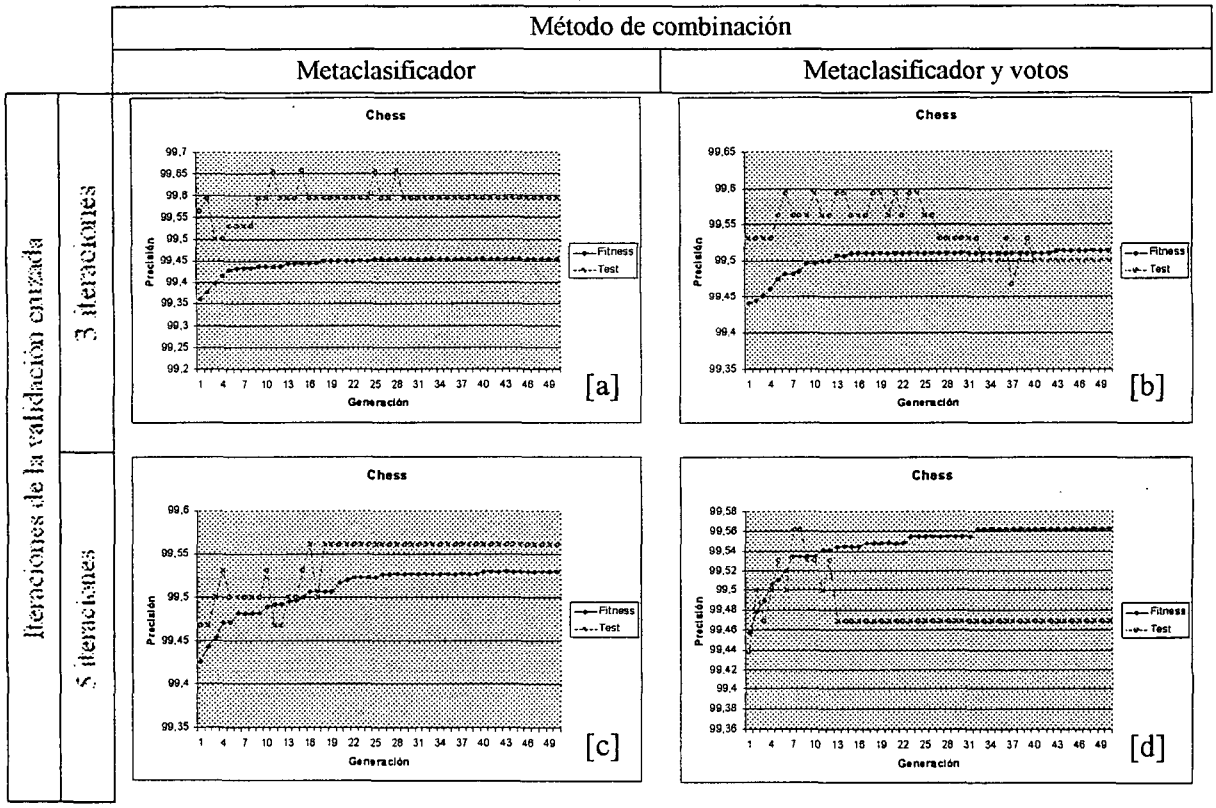


Figura 5.8: Resultados para el dominio Chess

Para el dominio *Chess* (Figura 5.8) los resultados tienen unos valores situados entre el 99,46% y el 99,59%. En este dominio la diferencia entre resultados obtenidos es mínima, todas las pruebas ofrecen buenos resultados, aunque la configuración que permite combinar únicamente por metaclasificador cuando el conjunto de entrenamiento se divide en 3 *folders* (Figura 5.8 [a]) es la que ha obtenido una mayor precisión. En este caso, destacan las pruebas en las que únicamente se permite la combinación por metaclasificador (Figura 5.8 [a] y Figura 5.8 [c]), puesto que el algoritmo en esos dos casos ha obtenido los mejores resultados. Para la configuraciones en las que se permite utilizar combinación por metaclasificador y por votos, se ha obtenido una precisión de 97,56% al dividir el conjunto de entrenamiento en 3 *folders* (Figura 5.8 [b]) y de 97,46% al dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.8 [d]).

Dominio *Glass*

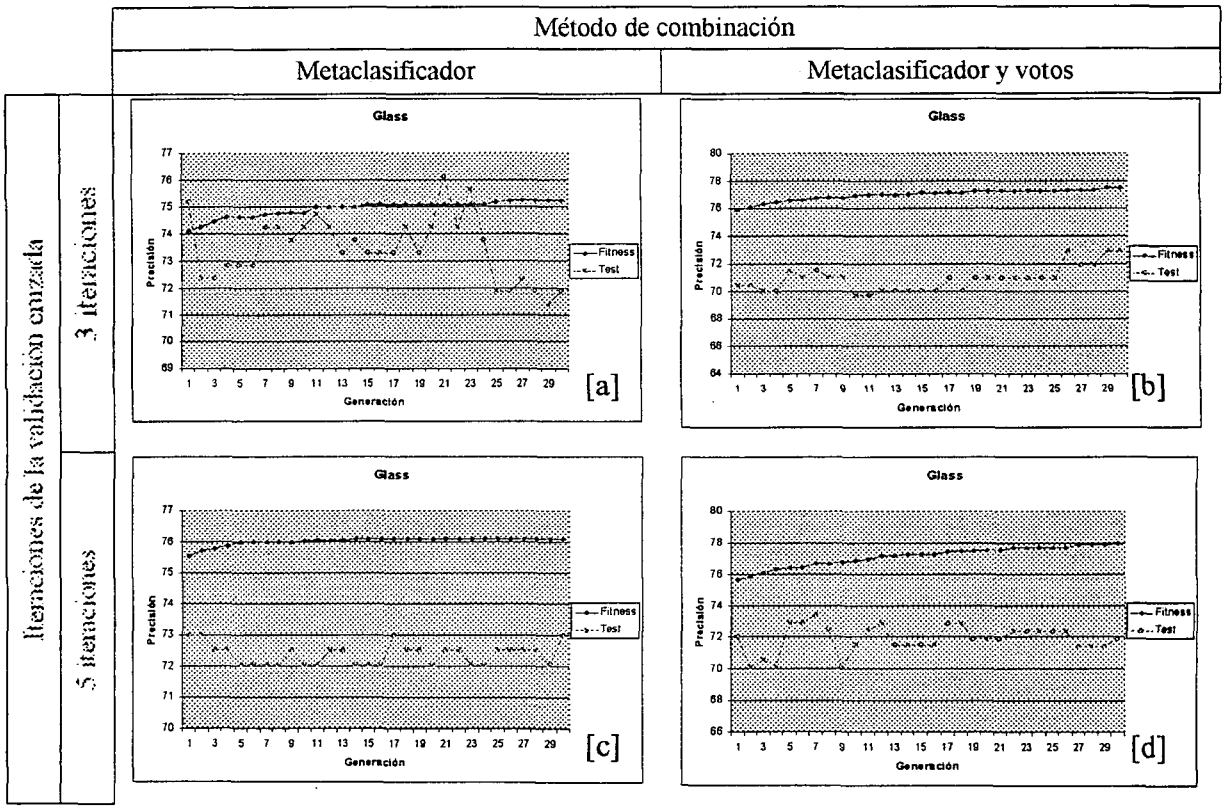


Figura 5.9: Resultados para el dominio Glass

Los resultados para el dominio *Glass* (Figura 5.9) tienen unos valores situados entre el 71,86% (Figura 5.9 [a]) y el 73% (Figura 5.9 [c]). La diferencia entre el mejor y el peor resultado es algo mayor en el dominio *Glass*, puesto que en este caso, esta diferencia es del 1.14% mientras que en otros dominios la diferencia es apenas del 0,02%. Los resultados que se obtienen en este dominio cuando únicamente se permite la combinación por metaclasificador son del 71,86% cuando el conjunto de entrenamiento se divide en 3 *folders* (Figura 5.9 [a]) y del 72,90% cuando el conjunto de entrenamiento se divide en 5 *folders* (Figura 5.9 [c]), mientras que cuando se utiliza combinación por metaclasificador y por votos se obtiene una precisión del 73% al dividir el conjunto de entrenamiento en 3 *folder*s (Figura 5.9 [b]) y del 71,90% al dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.9 [d])

Dominio *Hepatitis*

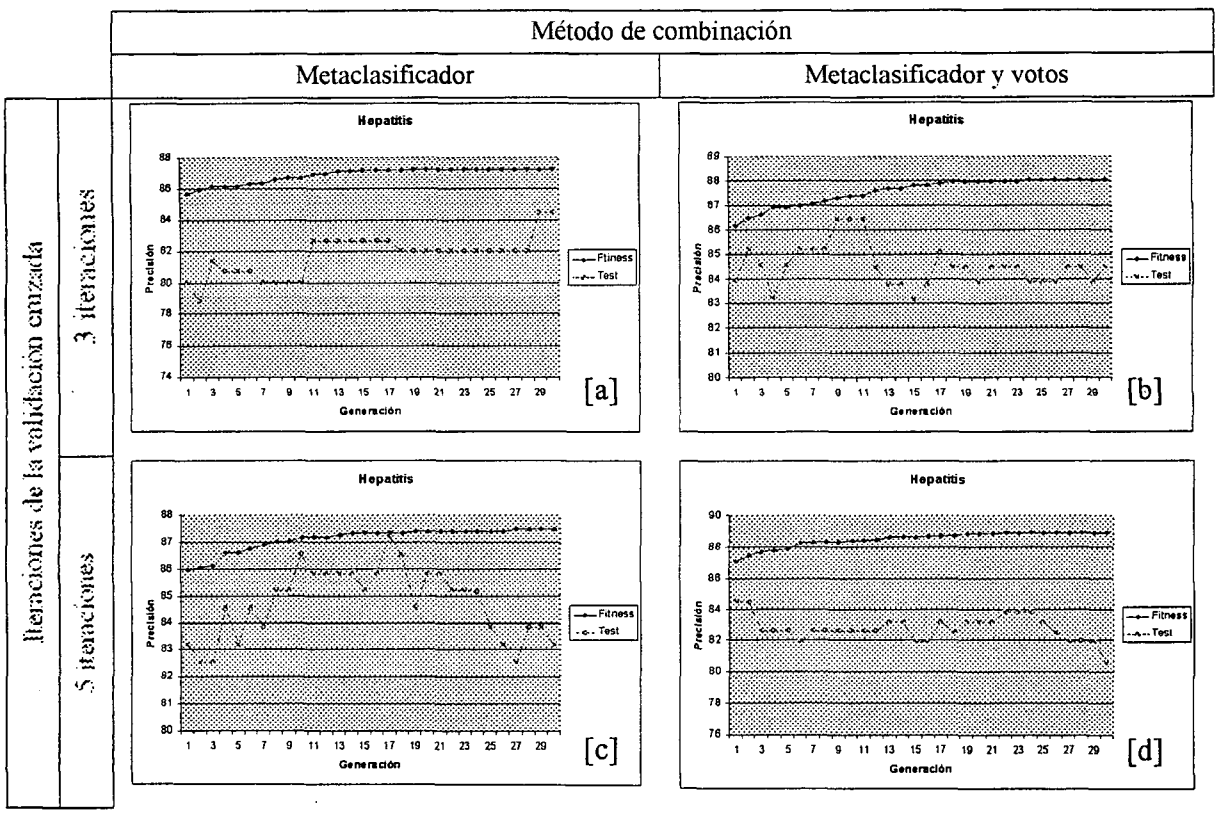


Figura 5.10: Resultados para el dominio Hepatitis

Los resultados que se obtienen para el dominio *Hepatitis* (Figura 5.10) cuando se utiliza combinación por metaclasificador y por votos son del 84,5% al dividir el conjunto de entrenamiento en 3 *folder*s (Figura 5.10 [b]) y del 80,58% al dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.10 [d]), mientras que cuando se utiliza combinación únicamente por metaclasificador la precisión es del 84,5% cuando el conjunto de entrenamiento se divide en 3 *folders* y cuando el conjunto de entrenamiento se divide en 5 *folders* (Figura 5.10 [c]). La diferencia entre el mejor y el peor resultado en este caso no es muy importante pero si reseñable, puesto que esta diferencia mayor del 1%. En este dominio los mejores resultados se obtienen cuando únicamente se permite la combinación por metaclasificador.

Dominio *Hypo*

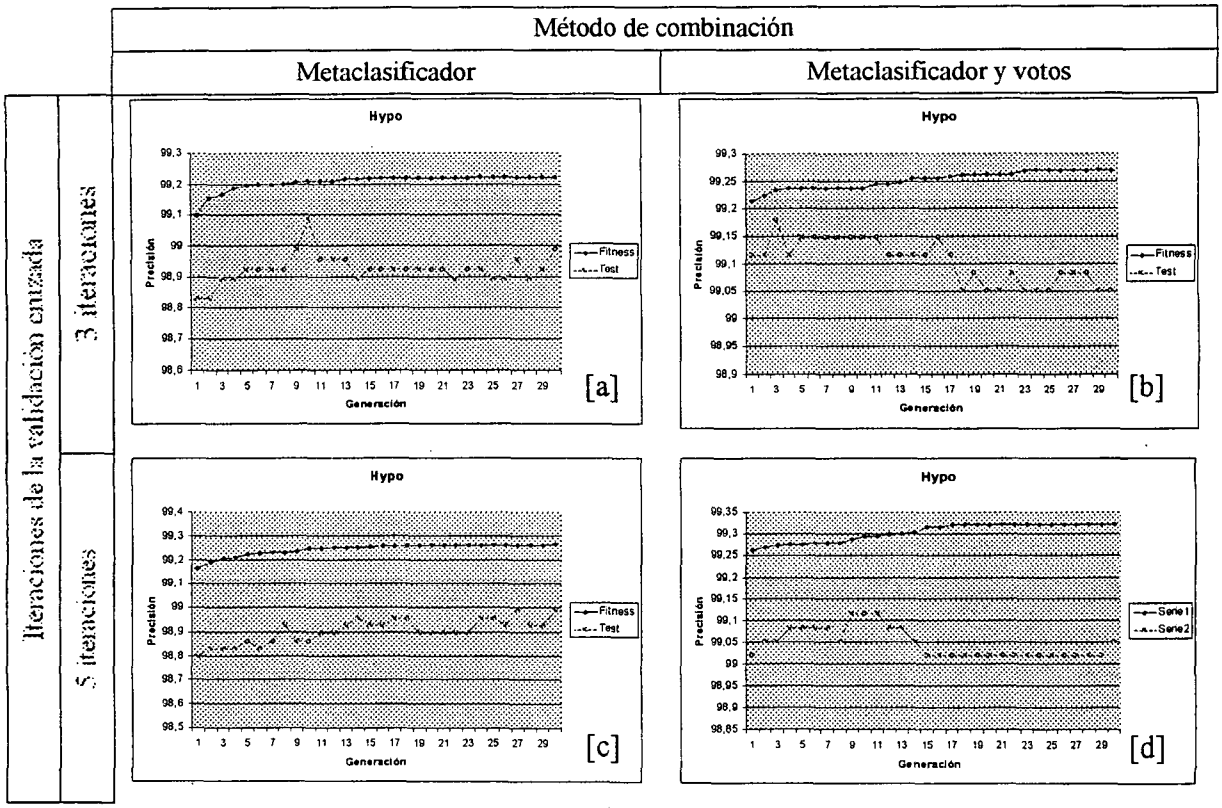


Figura 5.11: Resultados para el dominio Hypo

Los resultados para el dominio *Hypo* (Figura 5.11) tienen unos valores situados entre el 98,98% (Figura 5.11 [a] y Figura 5.11 [c]) y el 99,05% (Figura 5.11 [b] y Figura 5.11 [d]). Los resultados que se obtienen en este dominio cuando únicamente se permite la combinación por metaclasificador son del 98,98%, mientras que cuando se utiliza combinación por metaclasificador y por votos se obtiene una precisión del 99,05%. Según parece el factor que principalmente influye en el dominio *Hypo* es el método de combinación, pues el número de *folders* en los que se divide el conjunto de entrenamiento no ha influido en estos resultados.

5.5.3 Interpretación de los resultados

Según se observa en los resultados mostrados anteriormente, no todos los dominios responden del mismo modo a las diferentes configuraciones de GA-Ensemble. Se dan casos en los cuales el número de *folders* del conjunto de entrenamiento hace variar notablemente la precisión del algoritmo. Normalmente a mayor segmentación del conjunto de entrenamiento mayor precisión, pero también mayor tiempo de ejecución del algoritmo. Considerando la experimentación realizada se estimó que las pruebas que se han mostrado en este apartado, aquellas en la que la división del conjunto de entrenamiento se hace en tres y cinco *folders*, es cuando se obtiene la mejor relación precisión/tiempo de ejecución.

Respecto a la precisión y según se puede observar en la Figura 5.12, cuando únicamente se permite la combinación por metaclasificador se obtiene una precisión media del 87,58% en el caso de que el conjunto de entrenamiento se divide en 3 *folders* (Figura 5.12 [a]) y una precisión media del 87,83% en el caso de que el conjunto de entrenamiento se divide en 5 *folders* (Figura 5.12 [c]), mientras que cuando se utiliza combinación por metaclasificador y por votos se obtiene una precisión media del 88,18% al dividir el conjunto de entrenamiento en 3 *folder*s (Figura 5.12 [b]) y una precisión media del 87,07% al dividir el conjunto de entrenamiento en 5 *folders* (Figura 5.12 [d]).

Con los resultados promediados se observa como no existe una configuración que sea significativamente superior a la demás. Sin embargo los mejores resultados se obtienen cuando el método de combinación es mediante metaclasificador y votos, y el conjunto de entrenamiento se divide en 3 *folders*.

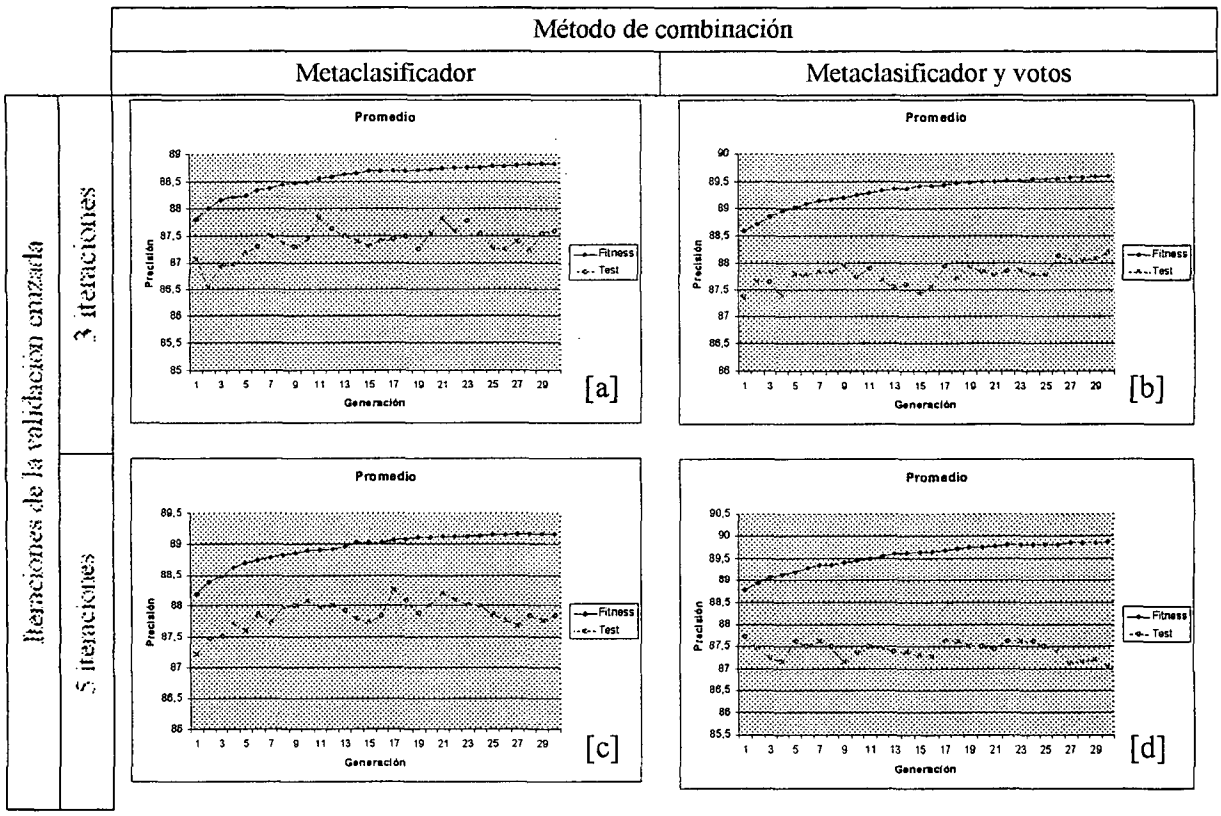


Figura 5.12: Resultados promediados para todos los dominios

5.5.4 Evaluación de los resultados

Si se atiende únicamente al porcentaje de acierto obtenido por el algoritmo en cada uno de los dominios, se puede observar como en todos los casos ha sido bastante bueno, obteniendo mayoritariamente tasas superiores al 75%.

Los resultados anteriores por sí solos no aportan la suficiente información, por lo que un buen modo de evaluar el desempeño del algoritmo es comparándolo con otros métodos de construcción de conjuntos homogéneos existentes, para poder contrastar el porcentaje de acierto del algoritmo en los dominios de la experimentación. Para realizar

la comparación se utiliza la configuración de *GA-Ensemble* en la que el método de combinación es mediante metaclasificador y votos, y el conjunto de entrenamiento se divide en 3 *folders*, porque es la configuración que, de media, mejores resultados ha obtenido (Figura 5.12 [b]). Los métodos de generación de conjuntos con los que se compara *GA-Ensemble* son:

- *Bagging* [3]: Método de construcción de conjuntos homogéneos basado en el submuestreo del conjunto de datos. Como clasificador base se ha utilizado *C4.5*

- *Boosting* [16]: Método de construcción de conjuntos homogéneos basado en la asignación de pesos a las instancias del conjunto de datos. La implementación utilizada es la del algoritmo *AdaBoostM1*. Como clasificador base se ha utilizado C4.5.

- *StackingC* [32]: Método similar a *Stacking*, pero con un reducido conjunto de atributos de meta-nivel. Las pruebas se han llevado a cabo usando diferente número de clasificadores base, primero usando 3 clasificadores base y después aumentándolo a 6 clasificadores base. En el caso de los 3 clasificadores base, estos han sido *C4.5*, *IBk* y *NaiveBayes*. Para las pruebas con 6 clasificadores base se han añadido *K**, *DecisionTable* y *ClassificationViaRegression*.

Dominio	GA-E	Boosting	Bagging	StC	
				3-BLC	6-BLC
australian	**86.66**	85.94	86.37	85.36	86.52
car	**97.68**	95.89	92.59	92.88	96.81
chess	99.56	**99.68**	99.40	99.40	99.40
diabetes	**76.81**	72.77	76.04	76.43	75.90
glass	72.90	73.33	71.51	68.70	**76.16**
hepatitis	84.50	80.54	79.95	84.54	**84.58**
hypo	99.05	99.05	99.11	**99.17**	99.11

Tabla 5.2: Comparativa de los resultados de GA-Ensemble

Para poder comparar los resultados, con estos algoritmos se aplicó una validación cruzada estratificada de 10 *folders*, al igual que con *GA-Ensemble*.

Se puede apreciar en la tabla 5.2 que en algunos dominios el algoritmo diseñado es mejor que los otros métodos de construcción de conjuntos homogéneos, no siendo significativamente peor en ningún caso.

En tres de los siete dominios empleados, *GA-Ensemble* obtiene los mejores resultados, aunque las diferencias con los otros sistemas que también obtuvieron buenos resultados no son significativas estadísticamente.

Si sumamos las diferencias de precisión entre *GA-Ensemble* y los otros clasificadores, podemos observar que *GA-Ensemble* obtiene una mejora relativa del 9,96% respecto de *Boosting*, 12,19% respecto de *Bagging* y 10,68% respecto de *StackingC* usando 3 clasificadores base. El único clasificador que obtiene mejores resultados que *GA-Ensemble* es *StackingC* usando 6 clasificadores base, este clasificador obtiene una mejora relativa del 1,32% respecto de *GA-Ensemble*. Como se puede observar, en menor o mayor grado *GA-Ensemble* supera a la mayoría de métodos comparados.

5.5.5 Eficiencia

Además de evaluar la precisión obtenida por *GA-Ensemble*, uno de los objetivos de este proyecto es mejorar la eficiencia de *GA-Stacking*, tal y como se expuso en el Capitulo 3, tratando de reducir al máximo la cantidad de recursos necesarios por el algoritmo para llevar a cabo el proceso de aprendizaje. Es por esto que uno de los puntos clave de este algoritmo es la creación del *pool* de clasificadores (Figura 4.2 [b]).

GA-Ensemble requiere un tiempo mucho menor que *GA-Stacking* para finalizar la ejecución del algoritmo genético. *GA-Ensemble* debe entrenar los clasificadores base una vez por cada iteración de la validación cruzada del conjunto de entrenamiento, mientras que *GA-Stacking* debe entrenar diversos clasificadores para poder calcular la función de *fitness* de cada individuo de la población. La mejora de *GA-Ensemble* en

tiempo de ejecución es obvia, se ve como *GA-Ensemble* requiere de un tiempo mucho menor que *GA-Stacking* para finalizar la ejecución del algoritmo genético

El tiempo requerido por *GA-Stacking* para terminar la ejecución del algoritmo genético se podría expresar según la formula:

$$T_{GA\text{-}Stacking} = ((Tm_{CB} * Nm_{CB}) + Tm_{MC}) * N_i * N_g$$

en donde:

Tm_{CB} = Tiempo medio para entrenar un clasificador base.
Nm_{CB} = Número medio de clasificadores que codifica un individuo.
Tm_{MC} = Tiempo medio para entrenar un metaclasificador.
N_i = Número de individuos de la población.
N_g = Número de generaciones del algoritmo genético.

Mientras que el tiempo necesario para *GA-Ensemble* termine la ejecución del algoritmo genético se podría expresar según la formula:

$$T_{GA\text{-}Ensemble} = (Tm_{CB} * Nt_{CB}) + ((Tm_{MC} * PI_m) * N_i * N_g)$$

en donde:

Tm_{CB} = Tiempo medio para entrenar un clasificador base.
Nt_{CB} = Número total de clasificadores base (clasificadores en el *pool*).
Tm_{MC} = Tiempo medio para entrenar un metaclasificador.
PI_m = Promedio de individuos en la población que usan el método del metaclasifcador
N_i = Número de individuos de la población.
N_g = Número de generaciones del algoritmo genético.

Como se ve, lo que realmente marca la diferencia entre los tiempos de *GA-Ensemble* y *GA-Stacking*, es que *GA-Ensemble* entrena los clasificadores base solo una vez por cada ejecución del AG, mientras que *GA-Stacking* lo realiza para evaluar cada individuo de la población.

Siguiendo la evaluación de la eficiencia de GA-Ensemble, a continuación se mostrarán los tiempos de ejecución sobre los diferentes dominios de la experimentación. En todas las pruebas realizadas se ha llevado un cálculo del tiempo necesario para la consecución de la ejecución. Absolutamente todas estas pruebas se han realizado sobre el mismo entorno, una única máquina con un núcleo de dos procesadores de 64-bits. Los tiempos de ejecución de las pruebas se van a mostrar para cada dominio individualmente para que así se tenga una visión más clara de los resultados.

	Metaclasificador		*Metaclasificador y votos*	
	3 iteraciones	*5 iteraciones*	*3 iteraciones*	*5 iteraciones*
diabetes	6 horas 9 minutos	12 horas 57 minutos	1 hora 25 minutos	2 horas 46 minutos
australian	3 horas 58 minutos	8 horas 5 minutos	1 hora 43 minutos	2 horas 13 minutos
car	26 horas 9 minutos	79 horas 22 minutos	18 horas 40 minutos	31 horas 28 minutos
chess	58 horas 53 minutos	88 horas 55 minutos	20 horas 41 minutos	26 horas 52 minutos
glass	9 horas 42 minutos	19 horas 20 minutos	3 horas 13 minutos	6 horas 47 minutos
hepatitis	1 hora 6 minutos	2 horas 17 minutos	0 horas 28 minutos	0 horas 37 minutos
hypo	26 horas 58 minutos	54 horas 36 minutos	10 horas 0 minutos	20 horas 50 minutos

Tabla 5.3: Tiempos de ejecución para todos los dominios

	Metaclasificador	*Metaclasificador y votos*
3 iteraciones	18 horas 59 minutos	8 horas 1 minuto
5 iteraciones	37 horas 46 minutos	13 horas 4 minutos

Tabla 5.4: Promedio de tiempos de ejecución

Tal y como se puede apreciar en las tabla 5.3 y 5.4, es bastante fácil ver como el tiempo de ejecución guarda una estrecha relación con el método de combinación y el número de iteraciones de la validación cruzada del conjunto de entrenamiento. El número de *folders* en la validación cruzada del conjunto de entrenamiento afecta en el sentido en que a mayor número de *folders* mayor número de veces debe entrenarme el *pool* de clasificadores, pues debe entrenarse una vez por cada *folder*. Lo que no es tan fácil de ver es como afecta el método de generación de conjuntos, ya que si se permite el uso de combinación por votos el tiempo se reduce considerablemente. Esto radica en el hecho de que en las soluciones en las que se usa la combinación por votos no hace falta entrenar un metaclasificador, con el ahorro de tiempo que eso supone.

La relevancia de estos resultados aparece cuando se comparan con los recursos necesitados por *GA-Stacking*, ya que, siendo ambos algoritmos similares en su estructura, los recursos necesarios son del orden de varias veces menores.

5.5.6 Descripción de las soluciones

Para analizar los resultados obtenidos por *GA-Ensemble* se han analizado los mejores individuos de cada una de los *folders* de la validación cruzada. Es decir, de las ejecuciones del AG llevadas a cabo en cada uno de los *folders* de la validación cruzada, se ha analizado el mejor de los individuos de la población.

En la Figura 5.13 se muestra la media del número de clasificadores base que poseen las soluciones obtenidas, de entre los 15 definidos en el apartado 5.2. Como se puede observar, el número máximo de clasificadores ha sido de 10, que es lo máximo que permite la configuración del algoritmo, pues en ese caso concreto dicho clasificador aparece en la solución de todos y cada uno de los *folders* de la validación cruzada.

En cuanto a los algoritmos utilizados para generar los clasificadores base se puede ver, como en cada dominio existen diversos algoritmos, normalmente cuatro o cinco, que están presentes en la mayoría de los *folders* de la validación cruzada.

En las figuras 5.13 y 5.14 se observan, para cada uno de los dominios, el número de *folders* de la validación cruzada en los que los algoritmos de aprendizaje se usan como parte del conjunto de clasificadores.

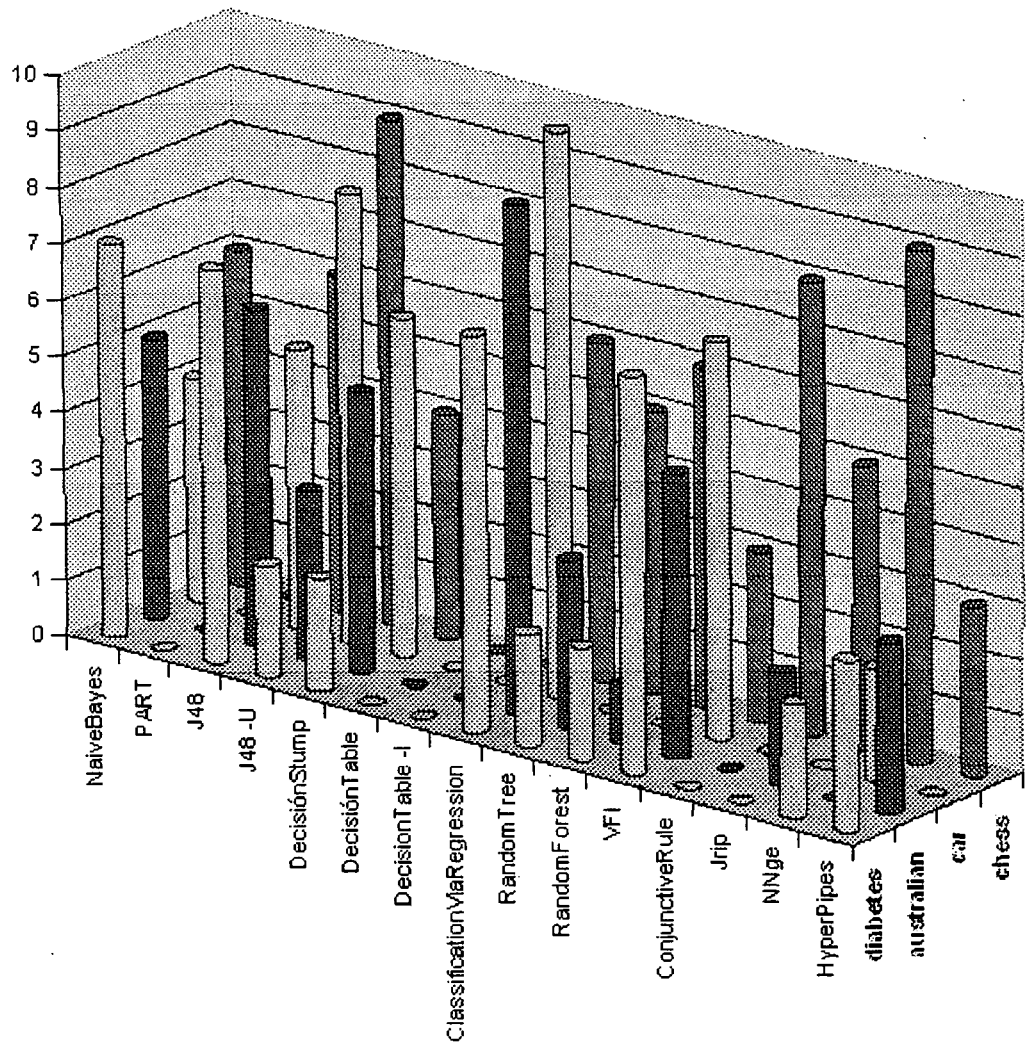


Figura 5.13: Número de folders de la validación cruzada en la que se usan los algoritmos en cada dominio.

Como se puede apreciar en las Figuras 5.12 y 5.13 los mejores individuos de cada iteración tienden a utilizar los mismos algoritmos para generar el conjunto de clasificadores. Por ejemplo, en el dominio *chess* se utilizan mayoritariamente los algoritmos *ConjunctiveRule*, *NNge* y *J48 –U*. También puede verse, como en otras ocasiones, es un algoritmo el que tiende a aparecer en la mayoría de los individuos de varios dominios, por ejemplo el algoritmo *ClassificationViaRegression* aparece como parte de la solución en muchos individuos de los dominios *diabetes, australian* y *car*.

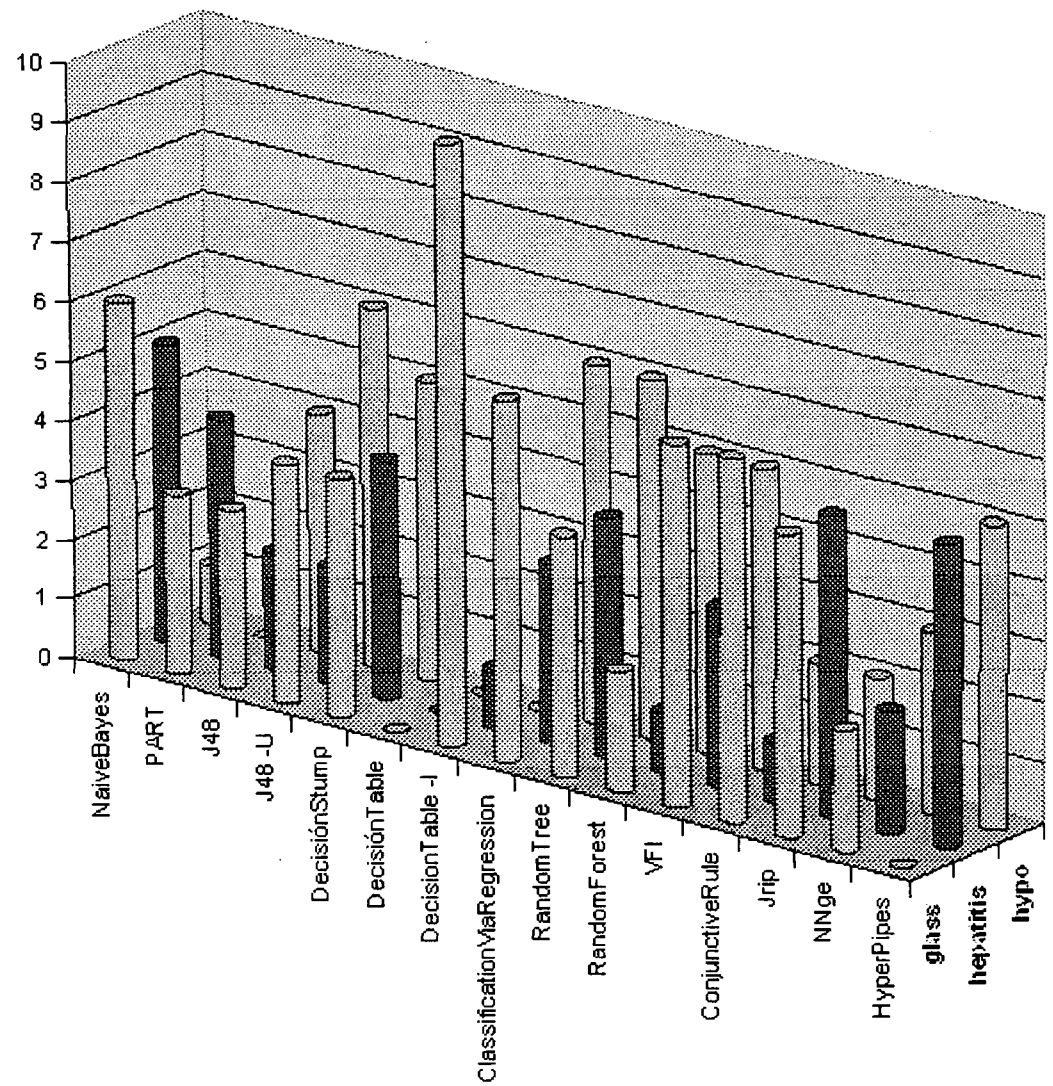


Figura 5.14: Número de folders de la validación cruzada en la que se usan los algoritmos en cada dominio.

En otros dominios, los individuos utilizan, prácticamente, todos los algoritmos disponibles en igual medida. Por ejemplo en el dominio *hepatitis,* la mayoría de los algoritmos son utilizados en un número muy similar.

Según se observa en las Figuras 5.13 y 5.14, qué y cuantos algoritmos tienen que ser utilizados depende en gran medida del dominio, por lo que *GA-Ensemble*, al no fijar una configuración "a priori", permite obtener conjuntos de clasificadores óptimos.

Capítulo 6

CONCLUSIONES

En este Proyecto Fin de Carrera se ha presentado un método que utiliza como base los algoritmos genéticos y un algoritmo de generación de conjuntos de clasificadores con la finalidad de obtener la mejor configuración de clasificadores para un dominio dado (*GA-Ensemble*). Uno de los problemas de los conjuntos de clasificadores es el que aborda este Proyecto Fin de Carrera, qué algoritmos utilizar para generar el conjunto (clasificadores base) y cómo combinarlos. Basándose en *GA-Stacking*, que utiliza algoritmos genéticos con el propósito de determinar la configuración óptima de *Stacking*, se lleva a cabo una búsqueda en el espacio de combinaciones de algoritmos y métodos de combinación con la finalidad de determinar el conjunto de clasificadores óptimo para un dominio dado.

Para mejorar la eficiencia de *GA-Stacking, GA-Ensemble* hace uso de un *pool* de clasificadores entrenados, o en otras palabras, entrena todos los algoritmos que vayan a tomar parte en la ejecución del algoritmo genético en una fase anterior a la búsqueda. De esta manera evita tener que entrenar los algoritmos de aprendizaje en sucesivas generaciones del algoritmo genético, tal y como hace *GA-Stacking*. Los algoritmos de aprendizaje que utiliza *GA-Ensemble* para generar los clasificadores son establecidos antes de iniciar el proceso de búsqueda.

GA-Ensemble divide el conjunto de datos del dominio mediante una validación cruzada, y divide, a su vez, los datos de entrenamiento de la primera validación cruzada mediante una segunda validación cruzada. Este proceso es necesario para poder obtener instancias de los datos con las que evaluar a los individuos de la población del algoritmo genético.

Además de los parámetros inherentes a los algoritmos genéticos, *GA-Ensemble* posee otros parámetros, de entre los que destacan dos especialmente relevantes: el número de *folders* en los que dividir el conjunto de datos de entrenamiento y el método de combinación de los clasificadores. Combinando estos dos parámetros se han evaluado cuatro configuraciones de *GA-Ensemble*. Analizando los resultados obtenidos por las configuraciones evaluadas se observa que el ampliar el espacio de búsqueda, no implica que se obtendrá una mejora significativa en los resultados. Por ejemplo, el aumentar el número de iteraciones de la validación cruzada de los datos de entrenamiento no supone una mejora significativa en la precisión del clasificador.

Para validar el método propuesto, se han realizado experimentos utilizando siete dominios para medir el rendimiento de *GA-Ensemble* frente a otros métodos de generación de conjuntos. Para comparar *GA-Ensemble* con los otros métodos de generación de conjuntos, se utilizó la configuración de *GA-Ensemble* que mejores resultados ofreció en la experimentación previa, aquella en la que se utilizan tres *folders* en la validación cruzada del conjunto de entrenamiento y se permite combinar los clasificadores mediante metaclasificador y mediante votos.

Los resultados empíricos demuestran que las soluciones encontradas por *GA.Ensemble* generan conjuntos de clasificadores que al ser comparados con métodos de generación de conjunto homogéneos, *Bagging* y *Boosting*, muestran mejores resultados. De igual forma, si se compara con un método de generación de conjuntos heterogéneos, *StackingC*, que utilice tres clasificadores base los resultados de *GA-Ensemble* siguen siendo mejores.

Por otro lado, al comparar los resultados de *GA-Ensemble* con *StackinC*, en el caso de que este use seis clasificadores base, los resultados no son tan buenos, pero en ningún caso significativamente peor.

La principal diferencia de *GA-Ensemble* con respecto a *GA-Stacking* es el uso del *pool* de clasificadores, lo que permite a *GA-Ensemble* mejorar su rendimiento. Comparando la eficiencia de *GA-Ensemble* respecto a *GA-Stacking*, se comprueba, tal y como se ha visto en el análisis, una mejora sustancial en el tiempo de ejecución del algoritmo. Esta mejora radica en el hecho de que *GA-Ensemble* necesita entrenar los clasificadores base solo una vez por cada ejecución del algoritmo genético, mientras que *GA-Stacking* lo realiza para evaluar cada individuo de la población.

Una de las principales ventajas de *GA-Ensemble* es su flexibilidad y extensibilidad, que le permite beneficiarse de los nuevos algoritmos de aprendizaje, ya que no esta restringido por ningún tipo de asunción "a priori". Otra ventaja de *GA-Ensemble* es que sus soluciones son independientes del dominio, por lo que es muy adaptable.

Sin embargo, aunque *GA-Ensemble* ha mejorado la eficiencia de *GA-Stacking*, todavía puede requerir un largo tiempo de ejecución dependiendo del dominio en el que se aplique. Aunque para la mayoría de los dominios, este inconveniente no es algo crucial, dado que las tareas de clasificación no suelen requerir trabajar en tiempo real.

Capítulo 7

TRABAJOS FUTUROS

En este capítulo se planten líneas de investigación que pueden ser estudiadas y desarrolladas en un futuro. Entre estas líneas se proponen las siguientes:

- Analizando los resultados obtenidos en la experimentación, se observa como la función de *fitness* mantiene un constante crecimiento mientras la precisión de los individuos sobre el conjunto de test varía. Se propone llevar a cabo un estudio con la finalidad de determinar el número adecuado de generaciones necesarias para lograr la precisión óptima sobre el conjunto de datos usado para validación.

- Las clases definidas en el paquete Weka tienen una estructura común, diseñada para obtener el máximo provecho de los diversos módulos de los que consta el paquete. Se propone modificar la implementación de la aplicación para que sea conforme al diseño que exige Weka, y así exista la posibilidad de incluirlo en el paquete como un nuevo algoritmo.

- Para los casos en los que varios clasificadores deban combinarse por medio de un metaclasificador, *GA-Ensemble* tiene definido un algoritmo por defecto. El utilizar diferentes metaclasificadores permitiría aumentar el

espacio de búsqueda del algoritmo. Se propone modificar la codificación de los cromosomas para que incluyan información del meta-nivel, y de este modo puedan usarse diferentes algoritmos para generar el clasificador que combine los clasificadores base.

- Al evaluar la función de *fitness*, *GA-Ensemble* sólo toma en cuenta la precisión del conjunto de clasificadores generado a partir del individuo. Si dos conjuntos poseen la misma precisión, ambos tienen el mismo *fitness*. Se plantea modificar la función de *fitness* para premiar a los individuos con menos clasificadores base y añadir presión para reducir el tamaño de los conjuntos, y de este modo ayudar a que primen las soluciones simples y precisas.

- En *GA-Ensemble*, los clasificadores base que componen un conjunto de clasificadores únicamente pueden combinarse utilizando un metaclasificador o a través de votos. Utilizar un número mayor de métodos de combinación aumentaría el espacio de búsqueda del algoritmo. Se plantea modificar la codificación de los cromosomas y los algoritmos de aprendizaje necesarios, para que se puedan utilizar nuevos métodos de combinación en la construcción de los conjuntos de clasificadores.

Bibliografía

[1] C. Blake and C. Merz. UCI repository of machine learning databases. Databases http://www.ics.uci.edu/mlearn/MLRepository.html, 1998.

[2] A.G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 1(72):81-138, 1995.

[3] L.Breiman. Bagging predictors. *Machine Learning*, 24(2):123-140, 1996.

[4]] E. Frank, Y. Wang, S. Inglis, G. Holmes, and I.H. Witten. Using model trees for classification. Machine Learning, 32(1):63-76, 1998.

[5] P. Chan and S. Stolfo. A comparative evaluation of voting and meta-learning on partitioned data. In M. Kaufman, editor, *Proceedings of Twelfth International Conference on Machine Learning*, pages 90-98, 1995.

[6] K.Cherkauer. Human expert-level performance on a scientific image analysis task by a system using combined artificial neural networks. En *Working Notes of the IAAA Workshop on Integrating Multiple Learned Models*, pages 15-21, 1996.

[7] William W. Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference*, 1995.

[8] Gulsen Demiroz and H. Altay Guvenir. Classification by voting feature intervals. In *Proceedings of the 9th European Conference on Machine Learning*, pages 85–92, 1997.

[9] A.P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likehood from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society B*, 1(38):1-38, 1977.

[10] T. G. Dietterich. Machine Learning research: four currents directions. *AI Magazine*, 18(4):97-136, 1997.

[11] T. G. Dietterich. Ensemble methods in Machine Learning. In J. Kittler and F. Roli, editors, *Multiple Classifiers Systems: first international workshop; proceedings /MCS 2000*, volume 1857 of *Lecture Notes in Computer Science*, pages 1-15, Cagliari, Italy, June 2000. Springer.

[12] T.G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263-286, 1995.

[13] E. Frank and I. Witten. *Data mining: practical machine learning tools and techniques with Java implementations.*Morgan Kaufmann, 2000.

[14] E. Frank and I. Witten. Generating accurate rule sets without global optimization. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 144-151. Morgan Kaufmann, 1998.

[15] Y. Freund y R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. En Springer-Verlag, editor, *Proceedings of the Second European Conference on Computational Learning Theory*, pages 23-37, 1995.

[16] Yoav Freund y Robert E. Schapire. Experiments with a new boosting algorithm. *Proc International Conference on Machine Learning*, pages 148-156, 1996.

[17] S. Haykin. *Neural networks; a comprehensive foundation*. Prentice Hall, 2nd edition, 1999.

[18] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[19] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 2ª edition, 1992.

[20] W. Iba and P. Langley. Induction of one-level decision trees. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 233-240. Morgan Kaufmann, 1992.

[21] G. John and P. Langley. Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338-345, 1995.

[22] R. Kohavi. The power of decision tables. In *Proceedings of the Eighth European Conference on Machine Learning*, 1995.

[23] Agapito Ledezma. *Aprendizaje automático en conjuntos de clasificadores heterogéneos y modelado de agentes*. PhD thesis, Universidad Carlos III de Madrid, 2004.

[24] Brent Martin. Instance-based learning: Nearest neighbor with generalization. Master's thesis, University of Waikato, 1995.

[25] P. Lanzi, W. Stolzmann, and S. Wilson, editors. *Learning Classifier Systems From Foundations to Applications*, volume 1813 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[26] S. P. Lloyd. Least squares quantization in PCM. In *IEEE Transactions on Infomration Theory*, number 28 in IT, pages 127-135, March 1982.

[27] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20, 1983.

[28] T.M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[29] J. Quinlan. Induction of decision trees. Machine Learning, 1(1):81-106, 1986.

[30] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[31] C. Schaffer. Cross-validation, stacking and bi-level stacking: Methods for classification learning. In P. Cheeseman and W. Oldford, editors, *Selecting models from data: Artificial Intelligence and Statistics IV*, pages 51-59. Springer-Verlag, 1994.

[32] A. K. Seewald. How to make stacking better and faster while also taking care of an unknown weakness. In A. G. H. Claude Sammut, editor, *Proceedings of the Nineteenth International Conference on Machine Learning (ICML 2002)*, Sidney, Australia, July 2002. Morgan Kaufmann.

[33] K. M. Ting. Decision combination based on the characterisation of predictive accuracy. *Intelligent Data Analysis*, 1(1-4):181-205, 1997.

[34] L. Todorovski and S. Dzeroski. Combining multiple models with meta decision trees. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 54-64, 2000.

[35] C. Watkins. *Learning from delayed Rewards*. PhD thesis, King's College. Cambridge, UK, 1989.

[36] I. Witten and E. Frank. *Data mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 2000.

[37] D. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[38] Z.H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: Many could be better than al. *Artificial Intelligence*, 137(1-2), 2002.

Apéndice A

MANUAL DE USUARIO

Este anexo se presenta un breve tutorial sobre el manejo de la aplicación, en él se incluyen instrucciones de cómo ejecutarla y configurar los parámetros de la misma.

A.1. Instalación

Tal y como se expuso en el apartado 2.6, este algoritmo utiliza Weka, el cual esta implementando en lenguaje Java y contiene numerosos algoritmos de aprendizaje automático para tareas de minería de datos. Igualmente, la implementación de *GA-Ensemble* se ha realizado utilizando el lenguaje orientado a objetos Java, ya que de este modo es independiente de la plataforma. Es por esto que el primer paso necesario para ejecutar la aplicación es tener instalada una Maquina Virtual de Java. La versión que se ha utilizado del paquete Weka es la 3.4.9, la cual necesita, como mínimo, disponer de la *Java Virtual Machine 1.4*, por ello antes de nada hay que asegurarse de tener instalado en el equipo el paquete Java J2SE 1.4 (descargable desde la página de SUN [1]).

Para poder ejecutar la aplicación también es necesario disponer del propio paquete de Weka, en su versión 3.4.9, el cual se puede obtener como un fichero tipo *jar* desde la página de los autores[2] .

[1] http://java.sun.com/
[2] http://www.cs.waikato.ac.nz/ml/weka/

El siguiente paso es incluir el *path* del paquete Weka en la variable *CLASSPATH* para que la maquina virtual lo pueda encontrar al compilar o ejecutar, este paso es ligeramente diferente dependiendo de la plataforma en la que se ejecute la aplicación. También se puede indicar el *CLASSPATH* explícitamente mediante una opción de línea de comandos: opción *–cp* o *–classpath* dependiendo de la plataforma.

Si la propia aplicación (el algoritmo *GA-Ensemble*) se encuentra contenido en un fichero tipo *jar* deberán seguirse los mismos pasos, añadiéndolo a la variable *CLASSPATH*.

Se recomienda también aumentar el número de memoria disponible para la maquina virtual, ya que por defecto la Maquina Virtual de Java solo tiene permitido el uso de cierta cantidad de memoria para ejecutar programas Java, y esta suele ser mucho menor que la cantidad de memoria RAM de la que dispone el ordenador. Se pueden aumentar la cantidad de memoria disponible para la máquina virtual usando las opciones apropiadas, por ejemplo

```
java – Xmx100m ...
```

para indicar que el tamaño máximo de la pila sea de 100 MB. Se recomienda aumentar este valor ya que el entrenamiento de los clasificadores requiere de una cantidad elevada de memoria. Para la experimentación se configuró que la maquina virtual pudiese 256MB, aunque en ocasiones este valor se aumentó llegando hasta los 512MB.

La interfaz de la aplicación no es gráfica, se ejecuta por línea de comandos, y los parámetros se suministran mediante un fichero diseñado para tal fin. La clase ejecutable de la aplicación es la que implementa la funcionalidad del algoritmo, es decir, la clase *Ensemble*. Una vez se han dado los pasos anteriores, teniendo instalada la maquina virtual y con la aplicación compilada (o en su defecto su fichero *jar)*, un ejemplo de cómo se ejecutaría la aplicación sería introduciendo en la línea de comandos.

```
java -Xmx512m -classpath .:weka.jar generator/Ensemble parametros.params
```

En el ejemplo se ha indicado el *CLASSPATH* con la opción *–classpath*, siendo en esta ocasión el *CLASSPATH* el directorio donde se ejecuta. El tamaño de la pila se ha aumentado a 512MB y se ha ejecutado la aplicación desde su directorio raíz, indicando que se quiere ejecutar la clase *Ensemble* (previamente compilada).

Tal y como se observa el único parámetro que recibe la aplicación es un fichero tipo *params,* en este fichero se definen todos los parámetros de la aplicación, es el único tipo de parámetro que puede recibir y es obligatorio. A continuación se detalla su contenido y como configurarlo.

A.2 Parámetros de la aplicación

La aplicación recibe un único fichero de parámetros en el que están incluidos todos los parámetros configurables de *GA-Ensemble*. Es un fichero de texto en el que puede figurar como mucho un parámetro por línea. Cada parámetro se identifica por el símbolo *@param* seguido del nombre del parámetro y de su valor entre llaves. Se permite que existan líneas en blanco, tabuladores y espacios, así como comentarios, los cuales se indican comenzado la línea con el símbolo %. Así, un ejemplo de la estructura de un fichero de parámetros sería la siguiente:

```
% Este el es primer comentario
@param parametro1 {valor1}

@param parametro2 {valor2}
@param parametro3{valor3}
%     Segundo comentario
```

Sin embargo la aplicación posee unos parámetros definidos, cada uno cumpliendo una determinada función. A continuación se definirá cada uno de los parámetros de la aplicación, viendo los valores que puede tomar y si es obligatorio o no, ya que mucho pueden tomar un valor por defecto.

arffFile – Indica el nombre del fichero de datos de entrada. Debe ser un fichero que tenga el mismo formato que los ficheros tipo *arff* que usa Weka. Este es un parámetro

obligatorio, pues son los datos del dominio a tratar. Los símbolos usados para el nombre deben de ser correctos para la plataforma donde se ejecute la aplicación. Su valor es una cadena. El fichero, por supuesto, debe existir.

Ejemplo de uso:

```
@param arffFile {hepatitis.arff}
```

outFile – Indica el nombre del fichero de datos de salida, en caso de que se configure la aplicación para que la salida no sea por pantalla. En él se almacenan los resultados obtenidos por el clasificador para cada iteración de la validación cruzada así como la precisión. Este no es un parámetro obligatorio, en caso de no introducirlo se usará como nombre del fichero *"default.out"*. Los símbolos utilizados para el nombre deben ser compatibles con la plataforma donde se ejecute la aplicación. Su valor es una cadena.

Ejemplo de uso:

```
@param outFile {hepatitis.out}
```

numFolds – Indica la cantidad de *folders* para la validación cruzada que se realizará sobre el conjunto de datos del dominio. Este no es un parámetro obligatorio, en caso de no introducirlo se aplicarán por defecto 10 *folders*. Su valor es un tipo entero

Ejemplo de uso:

```
@param numFolds {10}
```

numFoldsAg – Indica la cantidad de *folders* para la validación cruzada que se realizará sobre el conjunto de datos de entrenamiento. Este no es un parámetro obligatorio, en caso de no introducirlo se aplicarán por defecto 2 *folders*. Su valor es un tipo entero

Ejemplo de uso:

```
@param numFoldsAg {5}
```

seed – Indica la semilla para la generación aleatoria de números. Se utiliza para la estratificación del conjunto de datos. No es un parámetro obligatorio, en caso de no introducirlo se aplica por defecto la semilla 4576. Su valor es un tipo entero

Ejemplo de uso:

```
@param seed {130501}
```

classifiers – Indica los nombres de los algoritmos de aprendizaje que generarán los clasificadores del *pool*. Estos serán los clasificadores que se usen como clasificadores base para el conjunto generado. Se indican mediante el nombre de la clase del API de Weka, pueden incluir opciones y van separados por comas. No existe un límite máximo del número de clasificadores que se pueden utilizar. No es un parámetro obligatorio, en caso de no introducirlo se usan los 15 algoritmos de aprendizaje utilizados para la experimentación. Su valor es una cadena.

Ejemplo de uso:

```
@param classifiers {weka.classifiers.trees.J48,
weka.classifiers.lazy.IBk,
weka.classifiers.bayes.NaiveBayes,
weka.classifiers.trees.J48 -U}
```

numGenerations – Indica el número de generaciones que utilizará el algoritmo genético. No es un parámetro obligatorio, en caso de no introducirlo se aplica por defecto 100 generaciones. Su valor es un tipo entero

Ejemplo de uso:

```
@param numGenerations {50}
```

numCroms – Indica el número de individuos de la población del algoritmo genético. No es un parámetro obligatorio, en caso de no introducirlo se aplica por defecto 20 cromosomas. Su valor es un tipo entero

Ejemplo de uso:

```
@param numCroms {30}
```

eliteRate – Indica la tasa de elite de la población del algoritmo genético, esto es, el porcentaje de la población pasa a la siguiente generación sin modificarse. No es un parámetro obligatorio, en caso de no introducirlo se aplica por defecto una tasa de elite del 0.05 %. Su valor es un tipo real.

Ejemplo de uso:

```
@param eliteRate {0.10}
```

cullRate – Indica la tasa de desecho de la población del algoritmo genético. No es un parámetro obligatorio, en caso de no introducirlo se aplica por defecto una tasa de desecho del 0.05 %. Su valor es un tipo real.

Ejemplo de uso:

```
@param cullRate {0.10}
```

verbosity – Parámetro binario que especifica la salida de la aplicación. Indica si se desea que los resultados se muestren por pantalla o por el contrario se escriben en el fichero que define el parámetro *outFile*. En caso de darle valor 1 o *true* la salida es muestra por pantalla, si por el contrario se le da valor 0 o *false* se escribe en el fichero. No es un parámetro obligatorio, en caso de no introducirlo se escribe en el fichero de salida, es decir, se le da valor 0. Su valor es un tipo binario.

Ejemplo de uso:

```
@param verbosity,{1}
```

agMeta – Parámetro binario que especifica el método de combinación que puede usar el algoritmo. Indica si se desea usar únicamente un metaclasificador para combinar los clasificadores del conjunto o por el contrario se puede usar un metaclasificador y combinación por votos. En caso de darle valor 1 o *true* se permite combinar usando metaclasificador y combinación por votos, si por el contrario se le da valor 0 o *false* se permite usar únicamente combinación por metaclasificador. No es un parámetro obligatorio, en caso de no introducirlo se usa únicamente combinación por metaclasificador, es decir, se le da valor 0. Su valor es un tipo binario

Ejemplo de uso:

```
@param agMeta {1}
```

Apéndice B

MANUAL DE REFERENCIA

La función de este anexo es proporcionar los detalles técnicos necesarios que permitan ampliar la aplicación por parte de cualquier desarrollador.

Tal y como se ha comentado en la memoria, para la implementación de esta aplicación se ha utilizado el lenguaje de programación orientado a objetos Java. Se ha considerado el uso de este lenguaje como la mejor opción debido a que permite un desarrollo sencillo y rápido y además proporciona una plataforma de desarrollo segura, abierta, robusta, viable y flexible. Una vez instalada la aplicación, al ejecutarse la salida que genera puede almacenarse en un fichero o mostrarse directamente por pantalla. Los datos que componen la salida son información sobre los individuos que han sido seleccionados en cada ejecución del AG en cada una de las iteraciones. Por otra parte se los individuos que representan las soluciones finales, su porcentaje de cierto sobre el conjunto de datos de *fitness* y el conjunto de datos de test.

A continuación se incluye una descripción más detallada de las clases que componen la aplicación, en la que se incluyen sus atributos y operaciones más relevantes. El código fuente de dichas clases se puede encontrar en el soporte electrónico adjunto.

Clase My_output	
Descripción	Clase que define el *stream* o flujo de salida
Atributos	- *m_Verbosity* – Almacena el valor del parámetro *verbosity* (apartado 4.5.2). Indica si se desea que los resultados se muestren por pantalla o por el contrario se escriben en el fichero.
Operaciones	- *My_Output*- Constructor de la clase. Parámetros - *outFile* – nombre del fichero de salida - *verbosity* - valor del parámetro *verbosity* - *print* - Escribe una *línea* en el *stream* de salida. Parámetros - *theMsg* – cadena a escribir - *println* - *Escribe* una línea en el *stream* de salida incluyendo un salto de línea. Parámetros - *theMsg* – cadena a escribir

Clase Clock	
Descripción	Clase útil para calcular el tiempo de ejecución de la aplicación
Operaciones	- *setInicio* - Establece el tiempo de inicio. - *setFin* - Establece el tiempo de fin. - *elapsedTime* - Halla la diferencia entre los tiempos de inicio y fin para calcular el tiempo de ejecución y lo escribe por el *stream* de salida de la aplicación. Parámetros - *out* – stream o flujo de salida

Clase ParamsDataBase	
Descripción	Almacena los valores de todos los parámetros introducidos a la aplicación. Posee un atributo para cada parámetro (posibles parámetros definidos en el anexo A.2). Para aquellos parámetros no obligatorios, posee un valor predefinido.

Clase Param	
Descripción	Almacena el valor de un parámetro de la aplicación
Atributos	- *values* – Valores del parámetro, es un vector. Se considera que un parámetro puede tener varios valores para que puedan definirse parámetros como *classifiers*.
Operaciones	- *addValue* - Añade un valor al parámetro.

Clase ParamsReader	
Descripción	Lee el fichero de parámetros y comprueba que esta libre de errores. Para cada parámetro comprueba que esta definido de forma correcta y obtiene su valor. Genera un objeto de la clase *ParamsDataBase* que es utilizado por la aplicación posteriormente.
Operaciones	- *createParamsDataBase* - Crea la base de datos de los parámetros. Comprueba uno a uno los parámetros leídos en el fichero. Si el parámetro esta bien definido lo almacena. Retorno – objeto de la clase *ParamsDataBase*

Clase ClassifiersPool	
Descripción	Define el vector de clasificadores que conforma el *pool* de clasificadores entrenados
Operaciones	- *addClassifier- Dado un algoritmo de aprendizaje, añade un nuevo clasificador al pool de clasificadores.* Parámetros

- *name – nombre de la clase del API de Weka que define el algoritmo de aprendizaje*
- *options – array de opciones para el clasificador*

- *correct-* Devuelve el número de instancias bien clasificadas por un clasificador concreto del vector

  Parámetros
  - *index – posición del clasificador a usar dentro del vector de clasificadores que conforma el pool*
  - *m_Test – conjunto de instancias que usará en el test*

  Retorno – número de instancias bien clasificadas

- *pctCorrect-* Devuelve el porcentaje de instancias bien clasificadas por un clasificador concreto del vector

  Parámetros
  - *index – posición del clasificador a usar dentro del vector de clasificadores que conforma el pool*
  - *m_Test – conjunto de instancias que usará en el test*

  Retorno – porcentaje de instancias bien clasificadas

- *userPool* – Crea el *pool* de clasificadores según los algoritmos de aprendizaje introducidos como parámetros. Va tomando uno a uno los nombres de los algoritmos con sus opciones y los va añadiendo al vector que define el *pool* de clasificadores

  Parámetros
  - *classifiers– vector con los nombres de los algoritmos que compondrán el pool de clasificadores*

- *defaultPool* – Crea el *pool* de clasificadores con los algoritmos de aprendizaje que tiene la aplicación por defecto, es decir, con los 15 algoritmos de aprendizaje utilizados para la experimentación.

Clase PoolData	
Descripción	Almacena los *pools* de clasificadores de lo que hace uso la aplicación. Por cada iteración de la validación cruzada del conjunto de entrenamiento se genera un *pool* diferente.
Atributos	- *m_Pool* – Objeto de la clase *ClassifiersPool*. Contiene todos los clasificadores entrenados. - *m_Train* – Conjunto de datos utilizado para el entrenamiento de los clasificadores. - *m_Fitness* – Conjunto de datos que será utilizado para validar cada uno de los clasificadores del pool.

Clase Genetico	
Descripción	Define la funcionalidad del algoritmo genético. El AG es configurado según los parámetros facilitados por el usuario.
Operaciones	- *evolve*- Inicia la ejecución del algoritmo genético con los parámetros que se le han facilitado. Genera la población inicial, aplica los operadores y evalúa los individuos. Dado un cromosoma, para calcular su *fitness* realiza el siguiente proceso: ▪ Decodifica el individuo, para saber la posición de los clasificadores que formarán parte del conjunto. ▪ Esas posiciones las aplica sobre los conjuntos de datos facilitados para obtener los clasificadores a combinar. Dispondrá de más o menos conjuntos o *pools* de clasificadores dependiendo del número de *folders* en los que se divida el conjunto de datos de entrenamiento. ▪ Cada una de esas selecciones de clasificadores las combina, y haciendo uso del conjunto de datos de *fitness* calcula su precisión. ▪ Hace una media de las diversas precisiones obtenidas, otra vez tantas como el número de *folders* en los que se divida el conjunto de datos de entrenamiento. Esa media es el *fitness* del

individuo.

En cada una de las generaciones, una vez evaluados todos los individuos de la población, almacena el mejor cromosoma en un vector.

Parámetros

- *out – stream* o flujo de salida

Retorno – vector con los mejores individuos de cada generación

- *calculaFitness* – Calcula el *fitness* de un determinado cromosoma o individuo. Selecciona los clasificadores del *pool* que codifica el individuo, los combina mediante el método elegido, evalúa el conjunto de clasificadores generado mediante el conjunto de datos de *fitness* y devuelve la precisión del conjunto.

  Parámetros

  - *chrom – el cromosoma o individuo a evaluar*
  - *m_Training - conjunto de datos con el que se ha entrenado el pool de clasificadores, necesario para realizar la evaluación*
  - *m_Test – conjunto de datos con el que se va a evaluar el conjunto de clasificadores que se genere al descodificar el individuo*
  - *pool – el conjunto de clasificadores previamente entrenado*
  - *AGMeta – indica el método de combinación de los clasificadores que codifica el individuo*

  Retorno – precisión del conjunto de clasificadores generado, es considerado el *fitness* del individuo

- *fitnessMeta* –Calcula el porcentaje de aciertos de un determinado conjunto de clasificadores cuyos clasificadores base han sido combinados por medio de un metaclasificador. Se usa para calcular el *fitness* de los individuos que codifican que se use metaclasificador.

  Parámetros

  - *m_Training - conjunto de datos con los que se han entrado los clasificadores base , necesario para realizar la evaluación*

- *m_Test – conjunto de datos con el que se va a evaluar el conjunto de clasificadores*
- *chromVector– vector con el conjunto de clasificadores previamente entrenados que se usarán para generar el conjunto*

Retorno – precisión del conjunto de clasificadores generado

- *fitnessMetaVotos* – Calcula el porcentaje de aciertos de un determinado conjunto de clasificadores cuyos clasificadores base han sido combinados por medio de votos. Se utilizada para calcular el *fitness* de los individuos que especifican la combinación por votos.

Parámetros

- *m_Training - conjunto de datos con los que se han entrado los clasificadores base , estos datos son necesarios para realizar la evaluación*
- *m_Test – conjunto de datos con el que se va a evaluar el conjunto de clasificadores*
- *chromVector– vector con el conjunto de clasificadores previamente entrenados que se usarán para generar el conjunto*

Retorno – precisión del conjunto de clasificadores generado

- *bestNumAg* – Realiza diversas ejecuciones del algoritmo genético y devuelve un vector con los individuos de la ejecución en la que mejores resultados se obtuvieron.

Parámetros

- *out – stream o flujo de salida*
- *m_NumAg – número de veces que se desea ejecuta el algoritmo genético.*

Retorno – vector con los mejores individuos de cada generación de la ejecución del AG en al cual se llegó a una mejor solución

Clase ChromVector	
Descripción	Almacena individuos de la población del algoritmo genético. Para cada individuo se almacena la codificación de su cromosoma y su *fitness*
Operaciones	- *getMedVector* – Calcula el valor medio de los valores de *fitness* de los individuos que contiene el vector. El valor obtenido lo envía al *stream* de salida. Parámetros - *out – stream o flujo de salida*

Clase My_Bagging	
Descripción	Define el método de combinación por votos para clasificadores base ya entrenados, basándose en el algoritmo *Bagging* implementado en Weka. Es necesario incluirle con el constructor el conjunto de clasificadores base y es obligatorio que estos se encuentren entrenados con un conjunto de instancias de un formato similar al que se desea clasificar
Operaciones	- *distributionForInstance* – Calcula las probabilidades de pertenencia a una clase para la instancia de test dada. Parámetros - *instancia* – la instancia a ser clasificada Retorno – distribución de probabilidad predicha para la clase - *buildClassifier* – Establece el conjunto de clasificadores. Parámetros - *classifiers* – un array de clasificadores, previamente entrenados, con todas las opciones definidas

Clase My_Stacking	
Descripción	Define el método de combinación por metaclasificador para clasificadores base ya entrenados. Basado en el algoritmo *Stacking* implementado en Weka. Es necesario incluirle con el constructor el

	conjunto de clasificadores base y es obligatorio que estos se encuentren entrenados con un conjunto de instancias de un formato similar al que se desea clasificar
Operaciones	- *setClassifiers* – Establece los posibles clasificadores base de entre los que poder elegir. Estos clasificadores deben estar previamente entrenados. Parámetros - *classifiers* – un array de clasificadores, previamente entrenados - *distributionForInstance* – Calcula las probabilidades de pertenencia a una clase para la instancia de test dada. Parámetros - *instancia* – la instancia a ser clasificada Retorno – distribución de probabilidad predicha para la clase - *buildClassifier* – Selecciona un clasificador del conjunto de clasificadores minimizando el error del conjunto de entrenamiento. Parámetros - *data* – los datos de entrenamiento que se usaran para generar el clasificador.

Clase Ensemble	
Descripción	Define la propia funcionalidad del algoritmo de optimización de conjuntos de clasificadores. Es una clase ejecutable
Operaciones	- *main* – Controla el flujo de la aplicación, pide al sistema los recursos que necesite, crea las instancias necesarias y ejecutar cualquier otro método necesario para completar la funcionalidad del algoritmo. Su puede funcionamiento definirse mediante el pseudocódigo mostrado en la Figura B.1.

Pseudocódigo: función "main"

```
▪ LeerParámetros
▪ AbrirFicheroDatosDominio
  Si atributoClase es nominal
     - Estratificar conjunto
  Fin-sin
▪ CrearFlujoSalida
▪ EstablecerInstanteInicial
▪ Crear VectorMejoresIndividuos
  Mientras queden folders de la validacion cruzada de DatosDominio
     - ObtenerDatosEntrenamiento
     - ObtenerDatosTest
     - CrearVectorPools
       Si atributoClase de DatosEntrenamiento es nominal
            • Estratificar conjunto
       Fin-sin
       Mientras queden folders de la validacion cruzada de
       DatosEntrenamiento
            • ObtenerDatosEntrenamientoT
            • ObtenerDatosFitnessT
            • CrearPool con DatosEntrenamientoT
            • CrearDatosPool
            • Añadir DatosPool a VectorPools
       Fin-mientras
     - EjecutarAlgoritmoGenetico
     - ObtenerMejoresIndividuos
     - Añadir a VectorMejoresIndividuos
     - CrearPool con DatosEntrenamiento
       Si MejoresIndividuos no es vacio
            • CrearVectorResultadosMejoresIndividuos
              Mientras queden individuos
                 o EvaluarMejoresIndividuos con DatosTest
                 o Añadir a VectorResultadosMejoresIndividuos
              Fin-mientras
       Fin-si
  Fin-mientras
▪ Enviar a FlujoSalida ResultadosMejoresIndividuos
▪ EstablecerInstanteFinal
▪ Enviar a FlujoSalida IntervaloTiempo (InstanteFinal -
  InstanteInicial)
▪ CerrarFlujoSalida
```

Figura B.1: Pseudocódigo de la función main